XPS-D

Universal High-Performance Motion Controller/Driver

ESP COMPATIBLE

Newport®

Features Manual

Original instructions.

Table of Contents

1.0 Introduction 1
1.1 Scope of the Manual 1
1.2 Prerequisite 1

2.0 XPS Architecture 2
2.1 Introduction 2
2.2 State Diagrams 2
2.3 Motion Groups 4
2.3.1 Specific SingleAxis Group Features 5
2.3.2 Specific Spindle Group Features 5
2.3.3 Specific XY Group Features 5
2.3.4 Specific XYZ Group Features 5
2.3.5 Specific MultipleAxes Features 5
2.4 Native Units 6

3.0 Motion 8
3.1 Motion Profiles 8
3.2 Home Search 10
3.3 Referencing State 13
3.3.1 Move on Sensor Eevents 14
3.3.2 Moves of Certain Displacements 15
3.3.3 Position Counter Resets 15
3.3.4 State Diagram 16
3.3.5 Example: MechanicalZeroAndIndexHomeSearch 16
3.4 Move 16
3.5 Motion Done 18
3.6 JOG 20
3.7 Master Slave 21
3.8 Analog Tracking 22
3.8.1 Analog Position Tracking 22
3.8.2 Analog Velocity Tracking 23

4.0 Trajectories 25
4.1 Line-Arc Trajectories 25
4.1.1 Trajectory Terminology 25
4.1.2 Trajectory Conventions 26
4.1.3 Geometric Conventions 26
4.1.4 Defining Line-Arc Trajectory Elements 26

4.1.5 Define Lines 27
4.1.6 Define Arcs 27
4.1.7 Trajectory File Description 28
4.1.8 Trajectory File Examples 28
4.1.9 Trajectory Verification and Execution 29
4.1.10 Examples of the Use of the Functions 30
4.2 Splines 31
4.2.1 Trajectory Terminology 31
4.2.2 Trajectory Conventions 31
4.2.3 Geometric Conventions 31
4.2.4 Catmull-Rom Interpolating Splines 31
4.2.5 Trajectory Elements Arc Length Calculation 32
4.2.6 Trajectory File Description 33
4.2.7 Trajectory File Example 33
4.2.8 Spline Trajectory Verification and Execution 35
4.2.9 Examples 36
4.3 PVT Trajectories 36
4.3.1 Trajectory Terminology 36
4.3.2 Trajectory Conventions 36
4.3.3 Geometric Conventions 37
4.3.4 PVT Interpolation 37
4.3.5 Influence of the Element Output Velocity to the Trajectory 38
4.3.6 Trajectory File Description 39
4.3.7 Trajectory File Example 40
4.3.8 PVT Trajectory Verification and Execution 41
4.3.9 Example with a MultpleAxes Group 42
4.4 PT Trajectrories 42
4.4.1 Trajectory Terminology 42
4.4.2 Trajectory Conventions 42
4.4.3 Geometric Conventions 43
4.4.4 PT Interpolation 43
4.4.5 Trajectory File Description 43
4.4.6 Trajectory File Example 45
4.4.7 PT Trajectory Verification and Execution 47
4.4.8 Example of how to use PVT functions 47

5.0 Emergency Brake and Emergency Stop Cases 48
5.1 Principle 48
5.2 Emergency Brake Cases 49
5.3 Emergency Stop Cases 50

6.0 Compensation 51
6.1 Definitions 51
6.2 Backlash Compensation 52

6.3 Linear Error Correction....53
6.4 Positioner Mapping....53
6.5 XY Mapping....55
6.5.1 Multiple XY Mappings in Series....57
6.6 XYZ Mapping....58

7.0 Event Triggers....64
7.1 Events....64
7.2 Actions....74
7.3 Functions....79
7.4 Examples....80

8.0 Data Gathering....86
8.1 Time-Based (Internal) Data Gathering....87
8.2 Event-Based (Internal) Data Gathering....90
8.3 Function-Based (Internal) Data Gathering....92
8.4 Trigger-Based (External) Data Gathering....93

9.0 Output Triggers....95
9.1 Triggers on Line-Arc Trajectories....95
9.2 Triggers on PVT Trajectories....97
9.3 Triggers on PT Trajectories....98
9.4 Distance, Time Spaced Pulses or AquadB Position Compare....99
9.4.1 Even Distance Spaced Pulses Position Compare....99
9.4.2 Compensated Position Compare....102
9.4.3 XPS System of Coordinates....102
9.4.4 Compensated Position compare signals definition....104
9.4.5 Compensated Position compare scanning process description....104
9.4.6 Compensated Position Compare Related Functions....105
9.4.7 Time Spaced Pulses (Time Flasher)....106
9.4.8 AquadB Signals on PCO Connector....109

10.0 Control Loops....111
10.1 XPS Servo Loops....111
10.1.1 Servo structure and Basics....111
10.1.2 XPS PIDFF Architecture....113
10.1.3 PID Corrector Architecture....114
10.1.4 Proportional Term....114
10.1.5 Derivative Term....115
10.1.6 Integral Term....115
10.1.7 Variable Gains....116
10.2 Filtering and Limitation....117
10.2.1 Current velocity and current acceleration....117
10.3 Feed Forward Loops and Servo Tuning....118

10.3.1 Corrector = PIDFFVelocity....118
10.3.2 Parameters....118
10.3.3 Basics....118
10.3.4 Methodology of Tuning PID's for PIDFFVelocity Corrector (DC motors with or without tachometer)....119
10.3.5 Corrector = PIDFFAcceleration....120
10.3.6 Parameters....120
10.3.7 Basics....120
10.3.8 Methodology of Tuning PID's for PIDFFAcceleration Corrector (direct drive DC motors)....121
10.3.9 Corrector = PIDDual FFVoltage....122
10.3.10 Parameters....122
10.3.11 Basics....123
10.3.12 Methodology of Tuning PID's for PIDDualFF Corrector (DC motors with tachometers)....123
10.3.13 Corrector = PIPosition....123
10.3.14 Parameters....123
10.3.15 Basics & Tuning....124

11.0 Analog Encoder Calibration....125
11.1 Analog Encoder Errors....125
11.2 Analog Encoder Compensation Feature....127
11.3 Calibration Procedure....128

12.0 Excitation Signal....130
12.1 Introduction....130
12.2 How to Use the Excitation-Signal Function....130
12.3 Group State Diagram....131
12.4 Function Description....131

13.0 Introduction to XPS Programming....132
13.1 TCL Generator....133
13.2 Running Processes in Parallel....135

Service Form....137

Universal High-Performance Motion Controller/Driver XPS-D Controller

1.0 Introduction

1.1 Scope of the Manual

The XPS is an extremely high-performance, easy to use, integrated motion controller/driver offering high-speed communication through 10/100/1000 Base-T Ethernet, outstanding trajectory accuracy and powerful programming functionality. It combines user-friendly web interfaces with advanced trajectory and synchronization features to precisely control from the most basic to the most complex motion sequences. Multiple digital and analog I/O's, triggers and supplemental encoder inputs provide users with additional data acquisition, synchronization and control features that can improve the most demanding motion applications.

To maximize the value of the XPS Controller/Driver system, it is important that users become thoroughly familiar with available documentation.

The present **XPS-D Features Manual** describes all the functions implemented in the standard controller. Additional **Feature sheets** are available for special customer functions

1.2 Prerequisite

It is mandatory that both **XPS-D Start-Up Manual** and **User Interface Manual** be thoroughly read and understood before going through this manual.

Particularily, appropriate driver cards must be installed, all stages must be connected and an Ethernet connection must be established between the computer and the controller, either directly or through a network.

2.0 XPS Architecture

2.1 Introduction

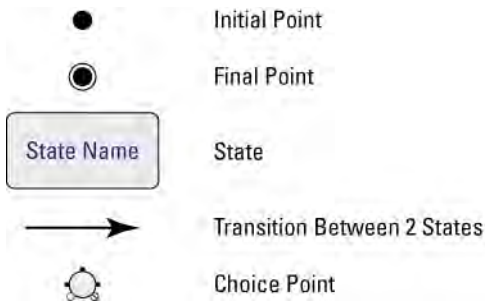
The architecture of the XPS firmware is based on an object-oriented approach. Objects are key to understanding this approach. Real-world objects share two characteristics: state and behavior. Software objects are modeled after real-world objects, so they have state and behavior too. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object implements its behavior with methods. A method is a function (subroutine) associated with an object. Therefore, an object is a software bundle of variables and related methods. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- **Modularity:** The source code for an object can be written and maintained independent of the source code for other objects. Also, an object can be easily passed around in the system.
- **Hidden information:** An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it.

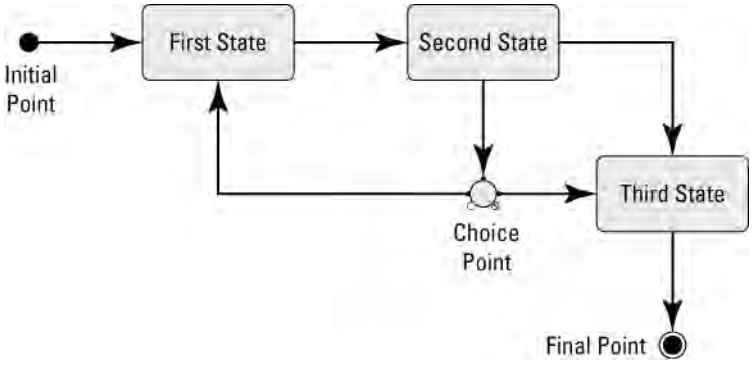
All objects have a life cycle and state diagrams are used to show the life cycle of the objects. The transition from one state to another is initiated after receiving a message from another object. Like all other diagrams, state diagrams can be nested in different layers to keep them simple and easy to read.

2.2 State Diagrams

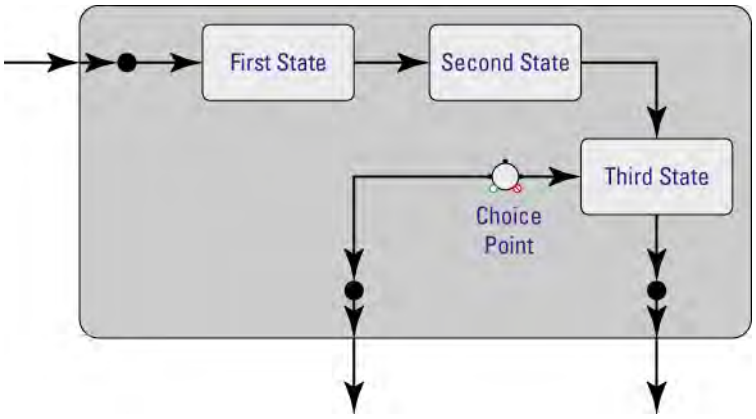
State diagrams are a way to describe the behavior of each group or object. They represent each steady state of a group and every transition between states in an exhaustive way. State diagrams contain the following components:



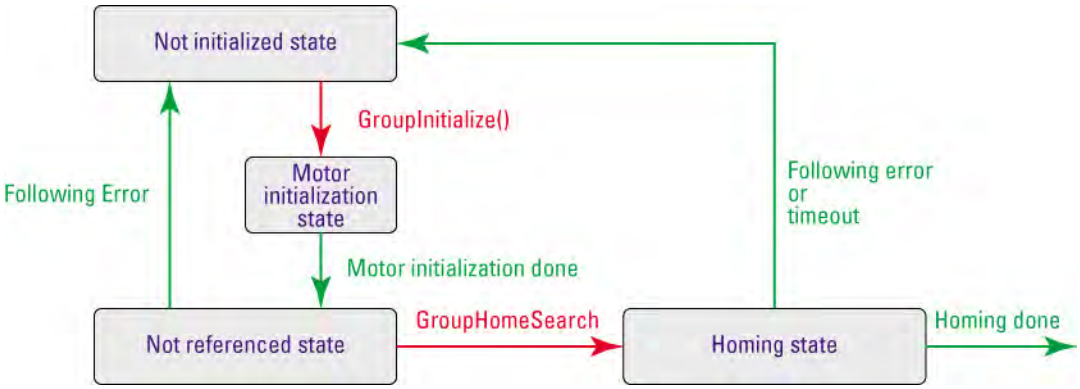
Here is an example of a simple stage diagram:



State diagrams can also include sub state diagrams:



The state diagrams that are specific to the XPS controller follow the same format. Within the XPS controller, all positioners are assigned to different motion groups. These motion groups have the following common state diagram:



As shown in the above state diagram, all groups have to be first initialized and then homed before any group is ready to perform any other function. Once the group is homed, it is in a ready state. There are five different motion groups available with the XPS controller:

- SingleAxis group
- Spindle group
- XY group
- XYZ group
- MultipleAxes group

Each group also has group specific states. Please refer to the Programmer's Manual for group-specific state diagrams for the five different groups.

All positioners of a group are bundled together for security handling. Security handling of different groups is treated independently. Following is a list of the different faults and consequences that can happen in the XPS controller:

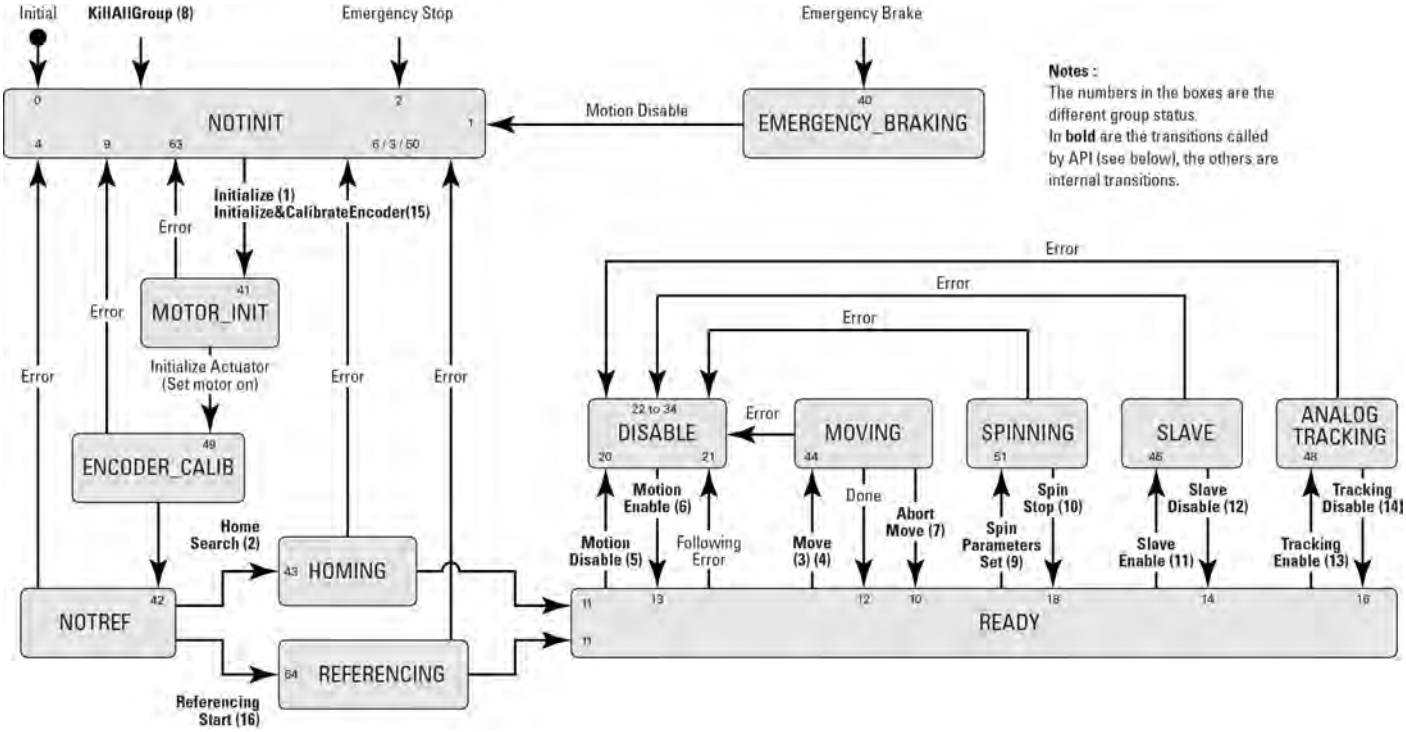
Error type	**Consequence**
General inhibition Motor fault Encoder fault	Emergency stop
End of travel	Emergency brake
Following error	Motion disable

- After an emergency brake or an emergency stop, both considered major faults, the corresponding group goes to a "not initialized" state: the system has to be initialized and homed again before any further motion.

- After a following error, as it is considered a minor fault, the corresponding group goes to a “Disable” state: a GroupMotionEnable() command puts the system back into “ready” state.

At any given time the group status can be queried from the controller. The function **GroupStatusGet (GroupName)** returns the current state number. The state numbers correspond to the state and to the event that generated the transition (if any). The function **GroupStatusStringGet (StateNumber)** returns the state description corresponding to the state number.

Similar to the Group State, the Controller Status can also be queried using the API **ControllerStatusGet**() or **ControllerStatusRead**(). The status numbers correspond to the status and event that generated the status. To get a description of the status code use **ControllerStatusStringGet**(). For more information refer to the XPS Programmer's Manual.



Called function

1. GroupInitialize	**7.** GroupMoveAbort	**13.** GroupAnalogTrackingModeEnable
2. GroupHomeSearch	**8.** GroupKill or KillAll	**14.** GroupAnalogTrackingModeDisable
3. GroupMoveAbsolute	**9.** GroupSpinParametersSet	**15.** GroupInitializeWithEncoderCalibration
4. GroupMoveRelative	**10.** GroupSpinModeStop	**16.** GroupReferencingStart
5. GroupMotionDisable	**11.** SpinSlaveModeEnable	**17.** GroupReferencingStop
6. GroupMotionEnable	**12.** SpinSlaveModeDisable	

State diagram of the XPS controller.

2.3 Motion Groups

Within the XPS controller, each positioner or axis of motion must be assigned to a motion group. This “group” can either be a SingleAxis group, a Spindle group, an XY group, an XYZ group or a MultipleAxes group. Once defined, the XPS automatically manages all safeties and trajectories of the motion group from the same function. For instance, the function **GroupHomeSearch (GroupName)** automatically homes the whole motion group GroupName independent of its definition as a SingleAxis group, a Spindle group, an XY group, an XYZ group or a MultipleAxes group. Within the system configuration file, system.ini, select the home sequence as “sequential”, one positioner after the other, or in “parallel”, with all positioners homing at the same time.

With a single function such as **GroupMoveAbsolute (GroupName, Position),** the whole motion group, GroupName, is moved synchronously to the defined absolute position, where **"Position"** may be one or more parameters depending on the number of positioners this motion group contains. This same command can be used to move a single positioner of a group to an absolute position by using the syntax **GroupMoveAbsolute (GroupName.PositionerName, Position1)**. These powerful, object-oriented functions are not only extremely intuitive and easy to use, they are also more consistent with other programming methods and reduce the number of commands learned compared to traditional mnemonic commands.

Another benefit provided by motion groups is improved error handling. For instance, whenever an error occurs due to a following error or a loss of the end-of-run signal, only the motion group where the error originated is affected (disabled) while all other motion groups remain active and enabled. The XPS manages these events automatically. This greatly reduces complexity and improves the security and safety of sensitive applications.

To illustrate this, let's consider a typical scanning application. If there is an error on the stepping axis of the XY table (which is set-up as an XY group), only the XY table is disabled while the auto-focusing tool (a vertical stage that is defined as a separate SingleAxis group) continues to function.

Each of the five available motion groups has specific features:

2.3.1 Specific SingleAxis Group Features

Master-Slave – To enable this function, the slaved positioner must be defined as a SingleAxis group. The master positioner can be a member of any motion group. So it is possible to define a Positioner as a slave of another positioner that is part of an XYZ group.

2.3.2 Specific Spindle Group Features

The Spindle Group is a single positioner group that enables continuous rotations with no limits and with a periodic position reset.

Master-Slave - In Master-Slave spindle mode the master and the slave group must be Spindle groups.

2.3.3 Specific XY Group Features

Line-Arc trajectories, XY mapping – These features are only available with XY groups. It is not possible for an XY group to perform a Spline or a PVT trajectory. Also, an XY group cannot be slaved to another group, however, any positioner of an XY group can be a master to a slaved SingleAxis group.

2.3.4 Specific XYZ Group Features

Spline trajectories, XYZ mapping – These features are only available with XYZ groups. It is not possible for an XYZ group to perform a Line-Arc or a PVT trajectory. Also, an XYZ group cannot be slaved to another group, however, any positioner of an XYZ group can be a master to a slaved SingleAxis group.

2.3.5 Specific MultipleAxes Features

PVT trajectories – PVT trajectories are only available with MultipleAxes groups. It is not possible for a MutipleAxes group to perform a Line-Arc or a Spline trajectory. Also, a MultipleAxes group cannot be slaved to another group. However, any positioner of a MultipleAxes group can be a master to a slaved SingleAxis group.

2.4 Native Units

The XPS controller supports user-defined native units like mm, inches, degrees or arcsecs. The units for each positioner are set in the configuration file where the parameter EncoderResolution indicates the number of units per encoder count. When using the XPS controller with Newport stages, this part of the configuration is done automatically. Once defined, all motions, speeds and accelerations can be commanded in the same native unit without any math needed. All other parameters like stage travel, maximum speed and all compensations are defined on the same scale as well. This is a great advantage compared to other controllers that can be commanded only in multiples of encoder counts, which can be an odd number.

In the XPS controller there are 4 types of position information for each positioner: TargetPosition, SetpointPosition, FollowingError and CurrentPosition. These are described as follows:

The **CurrentPosition** is the current physical position of the positioner. It is equal to the encoder position after all compensations (backlash, linear error and mapping) have been taken into account.

The **SetpointPosition** is the theoretical position commanded to the servo loop. It is the position where the positioner should be, during and after the end of the move.

The **FollowingError** is the difference between the CurrentPosition and the SetpointPosition.

The **TargetPosition** is the position where the positioner must be after the completion of a move.

When the controller receives a new motion command after the previous move is completed, a new TargetPosition is calculated.

This new target is received as an argument for absolute moves. For relative moves, the argument is the length of the move and the new target is calculated as the addition of the current target and the move length. Then the profiler of the XPS calculates a set of SetpointPositions to determine where the positioner should be at each given time.

When the positioner is controlled by a digital servo loop with a PID corrector, part of the signals sent to the motor of the positioner is a function of the following error. Part of this function is the integral gain of the PID filter that requires a following error equal to zero to reach a constant value.

The encoder in the positioner delivers a discrete signal (encoder counts). Take the example of an encoder with a resolution of 1 and a target position equal to 1.4. The real position cannot reach the value of the target position (1 or 2 instead of 1.4), so the following error will never be equal to zero (closest values are +0.6 and -0.4). Thus, due to the integral gain of the PID filter, the system will never settle, but will oscillate between the positions 1 and 2.

The XPS controller avoids this instability while allowing the use of native units instead of encoder counts by using a rounded value of the TargetPosition to calculate the motion profile and a rounded value for the following error. But the non-rounded value of the TargetPostion will be stored as final position, so that there is no accumulation of errors due to rounding, in case of successive relative moves.

To understand the difference, consider a positioner with a resolution of 1 that is at the position 0. This positioner receives a relative motion command of 10.4. At the end of the motion the CurrentPosition will be 10 and the SetpointPosition will be 10, but the TargetPosition will be 10.4. The positioner then receives the same relative motion command again. At the end of this motion the CurrentPosition will be 21, the SetpointPosition will be 21 and the TargetPosition will be 20.8.

NOTE

When an application requires a sequence of small incremental motion of constant step size close to the encoder resolution, make sure that the commanded incremental motion is equal to a multiple of encoder steps.

The TargetPosition, SetpointPosition, CurrentPositon and FollowingError can be queried from the controller using the appropriate function calls.

3.0 Motion

3.1 Motion Profiles

Motion commands refer to strings sent to a motion controller that will initiate a motion. The XPS controller provides several modes of positioning from simple point-to-point motion to the most complex trajectories. On execution of a motion command, the positioner moves from the current position to the desired destination. The exact trajectory for the motion is calculated by a motion profiler. So the motion profiler defines where each of the positioners should be at each point in time. There are details worth mentioning about the motion profiler in the XPS controller:

In a classical trapezoidal motion profiler (trapezoidal velocity profile), the acceleration is an abrupt change. This sudden change in acceleration can cause mechanical resonance in a dynamic system. In order to eliminate the high frequency portion of the excitation spectrum generated by a conventional trapezoidal velocity motion profile, the XPS controller uses a sophisticated SGamma motion profile. Figure 1 shows the acceleration, velocity and position plot for the SGamma profile.

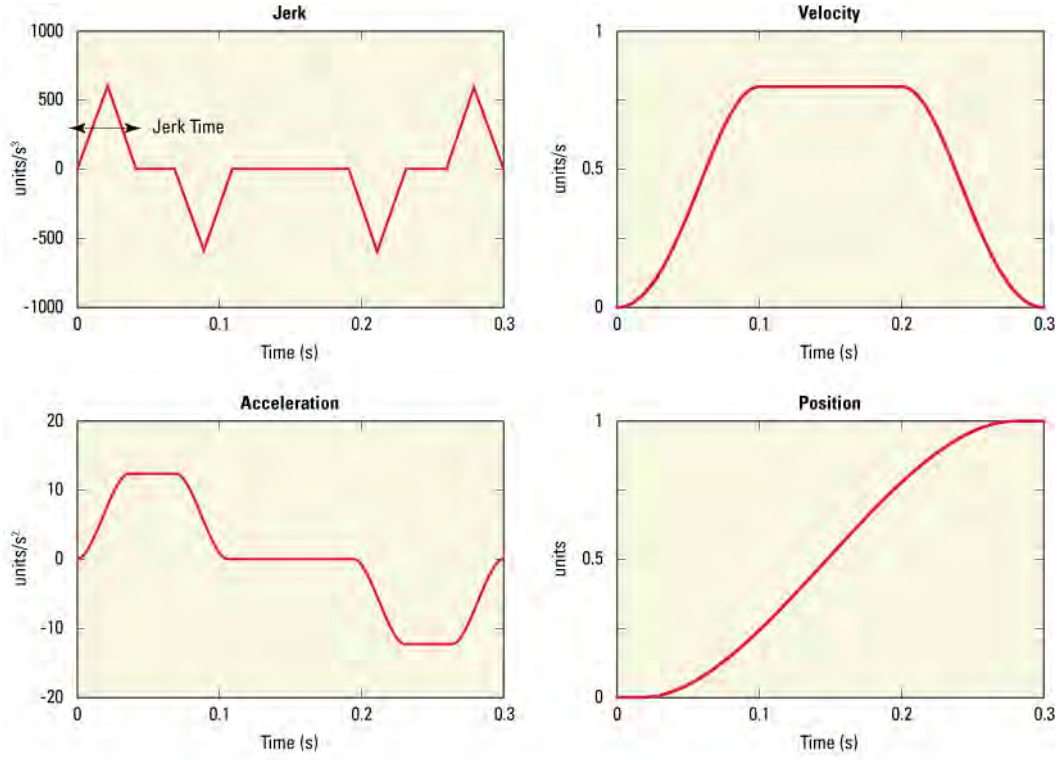


Figure 1: SGamma motion profile.

The SGamma motion profile provides better control of dynamic systems. It allows for perfect control of the excitation spectrum that a move generates. In a multi-axes system this profile gives better control of each axis independently, but also allows control of the cross-coupling that are induced by the combined motion of the axes. As shown in figure 17, the acceleration plot is parabolic. The parabola is controlled by the jerk time (jerk being the derivative of the acceleration). This parabolic characteristic of the acceleration results in a much smoother motion. The jerk time defines the time needed to reach the

necessary acceleration. One feature of the XPS controller is that it automatically adapts the jerk time to the step width by defining a minimum and a maximum jerk time. This auto-adaptation of the jerk time allows a perfect adjustment of the system's behavior with different motion step sizes.

NOTE

Because of jerk-controlled acceleration, any move has a duration of at least four times the jerk time.

For the XPS controller, the following parameters need to be configured for the SGamma profile:

- MaximumVelocity (units/s)
- MaximumAcceleration (units/s^2)
- EmergencyDecelerationMultiplier (Applies to Emergency Stop)
- MinimumJerkTime (s)
- MaximumJerkTime (s)

The above parameters are set in the stages.ini file for a positioner. When using the XPS controller with Newport stages, these parameters are automatically set during the configuration of the system.

The velocity, acceleration and jerk time parameters is modified by the function **PositionerSGammaParametersSet**().Note that for continuity reason, the effective maximum velocity of the motion must be adjusted and may not be exactly the value defined by the parameter MaximumVelocity. For motion where a very accurate value of the velocity is needed, the length of the displacement has to be adjusted to the value given by the API **SGammaExactVelocityAjustedDisplacementGet**.

Example

PositionerSGammaParametersSet (MyGroup.MyStage, 10, 80, 0.02, 0.02)

This function sets the positioner "MyStage" velocity to 10 units/s, acceleration to 80 units/s^2 and minimum and maximum jerk time to 0.02 seconds. The set velocity and acceleration must be less than the maximum values set in the stages.ini file. These parameters are not saved if the controller is shut down. After a re-boot of the controller, the parameters will retain the values set in the stages.ini file.

In actual use, the XPS places a priority on the displacement position value over the velocity value. To reach the exact position, the speed of the positioner may vary slightly from the value set in the stages.ini file or by the **PositionerSGammaParametersSet** function. So the drawback of the SGamma profile is that the velocity used during the move can be a little bit different from the velocity defined in the parameters. For example, the exact velocity will change when the move distance is changed, move 100 mm, then 100.001 mm then 100.011 mm. There will be some changes to the commanded velocity. This change can be ignored for many applications except where an accurate time synchronization during the motion is required.

The function, **PositionerSGammaExactVelocityAdjustedDisplacementGet**(), can be used as described below to achieve the exact desired speed in applications that require an accurate value of the velocity during a move. In this case, the velocity value is adhered to, but the target position may be slightly different from the one required. In other words, according to the application requirements, the user can choose between very accurate positions or very accurate velocities.

Example

PositionerSGammaExactVelocityAdjustedDisplacementGet (MyGroup.MyStage, 50.55, ExactDisplacement)

This function returns the exact displacement for that move with the exact constant velocity set shown in the example above (10 mm/s). The result is stored in the variable ExactDisplacement, for instance 50.552.

GroupMoveAbsolute (MyGroup.MyStage, 50.552)

In the above example, for a position of 50.55 mm, the command returns a value of 50.552. This means that in order for the positioner "MyStage" to achieve the desired velocity in the most accurate way, the commanded position should be 50.552 mm instead of 50.55 mm.

The XPS can report two different positions. The first one is the SetpointPosition or theoretical position. This is the position where the stage should be according to the profile generator.

The second position is the CurrentPosition. This is the actual position as reported by the positioner's encoder after taking into account all compensation. The relationship between the SetpointPosition and the CurrentPosition is as follows:

Following error = SetpointPosition - CurrentPosition

The functions to query the SetpointPosition and the CurrentPosition values are:

GroupPositionCurrentGet() **and GroupPositionSetpointGet**()

3.2 Home Search

Home search is a specific motion process. Its goal is to define a reference point along the course of travel accurately and repeatably. The need for this absolute reference point is twofold. First, in many applications, it is important to know the exact position in space, even after a power-off cycle. Secondly, to prevent the motion device from hitting a travel obstruction set by the application (or its own hardware travel limits), the controller uses software limits. To be efficient, the software limits must be referenced accurately to the home before running the application.

After motor initialization, any motion group must first be homed or referenced before any further motion can be executed. Here, homing refers to a predefined motion process that moves a stage to a unique reference position and defines this as Home. Referencing refers to a group state that allows the execution of different motions and the setting of the position counters to any value (see next section for details). The referencing state provides flexibility for the definition of custom home search and system recovery processes. It should only be used by experienced users.

A number of hardware solutions may be used to determine the position of a motion device, the most common are incremental encoders. By definition, these encoders can only measure relative position changes and not absolute positions. The controller keeps track of position changes by incrementing or decrementing a dedicated counter according to the information received from the encoder. Since there is no absolute position information, position "zero" is where the controller was powered on (and the position counter was reset).

To determine an absolute position from incremental encoders, the controller must use a reference position that is unique to the entire travel, called a home switch or origin switch, usually in conjunction with an index pulse.

An important requirement is that this switch must have the same resolution as the encoder pulses.

If the motion device uses a linear scale as a position encoder, the home switch is usually placed on the same scale and read with the same resolution.

If, on the other hand, a rotary encoder is used, homing becomes more complicated. To have the same resolution, a mark on the encoder disk could be used (called index pulse), but because the mark repeats every revolution, it does not define a unique point over the

entire travel. An origin switch, on the other hand, placed in the travel of the motion device is unique, but typically is not precise or repeatable enough. The solution is to use both in a dedicated search algorithm as follows.

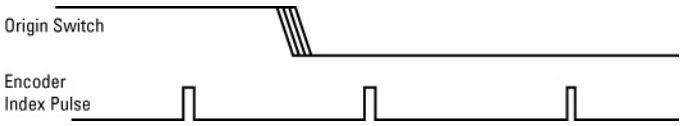


Figure 2: Home (Origin) switch and encoder index pulse.

A Home switch (Figure 2) separates the entire travel in two areas: one has a high level and the other has a low level. The most important part is the transition between the two areas. Just by looking at the origin switch level, the controller knows already on which side of the transition the positioner is and which direction to start the homing process.

The task of the home search process is to define one unique index pulse as the absolute position reference. This is first done by finding the home switch transition and then the very first index pulse (Figure 3).

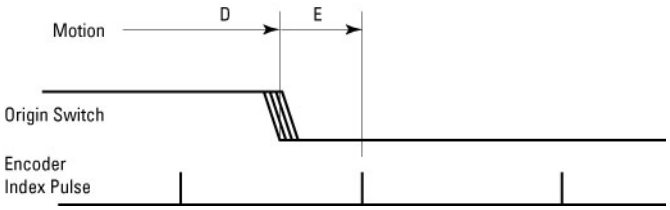


Figure 3: Slow-Speed Origin Switch Search.

Labeling the two motion segments D and E, the controller searches for the origin switch transition in D and for the index pulse in E. To guarantee the best repeatability possible, both D and E segments must perform at a very low speed and without stopping in between.

The homing process described above has a drawback. At low search speeds, the process could take a very long time if the positioner happens to start from the one end of travel. To speed things up, the positioner is moved fast until it is in the vicinity of the origin switch and then performs the two slow motions, D and E, at half the home search velocity. The new sequence is shown in Figure 4.

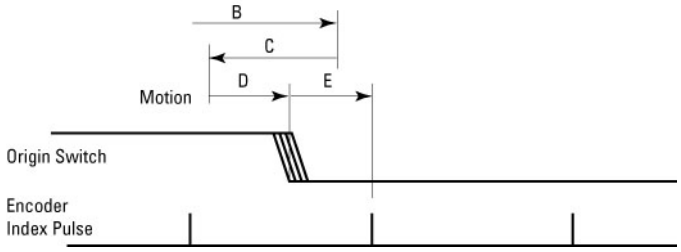


Figure 4: High/Low-Speed Home (Origin) Switch Search.

Motion segment B is performed at the pre-programmed home search speed. When the home switch transition is encountered, the motion device stops (with an overshoot), reverses direction and searches for the switch transition again, this time at half the speed (segment C). Once the switch transition is encountered, it stops again with an overshoot, reverses direction and executes D and E with one tenth of the programmed home search speed.

In the case when the positioner starts from the other end of the home switch transition, the routine is shown in Figure 5.

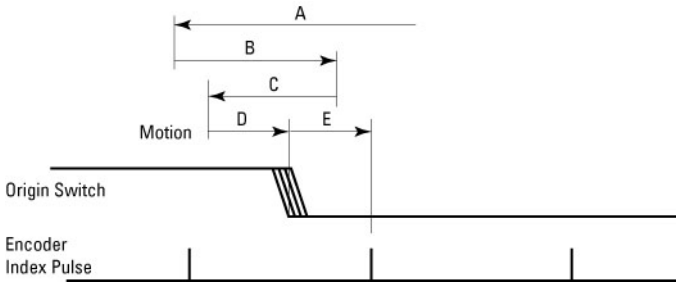


Figure 5: Home (Origin) Search from Opposite Direction.

The positioner moves at the home speed up to the home switch transition (segment A) and then executes segments B, C, D and E as in Figure 5.

This home search process guarantees that the last segment, E, is always performed in the positive direction of travel and at the same reduced speed. This method ensures an precise and repeatable reference position.

There are 7 different home search processes available in the XPS controller:

1. **MechnicalZeroAndIndexHomeSearch** is used when the positioner has a hardware home switch plus a zero index from the encoder. This process is the default for most Newport standard stages.
2. **MechanicalZeroHomeSearch** is used with positioners that have a hardware home switch but with no zero index from the encoder.
3. **IndexHomeSearch** is used with positioners that have a home index, but with no hardware home switch signal. In this process, the positioner initially moves in the positive direction to find the index. When a + limit switch is detected, the direction of motion reverses until the index is found.
4. **CurrentPositionAsHome** is used when the positioner has no home switch or index. This process will keep the positioner's home at its current location. Setting the home too close to the EOR could generate unwanted emergency stops. Start with around 50 MIM (Minimum Incremental Movement) units, but an optimum distance may be determined by trial and error, depending on the stage.
   This feature can also be used to set home arbitrarily and bypass a home switch.
5. **MinusEndOfRunAndIndexHomeSearch** uses the positioner's minus end-of-run limit as a hardware home switch and a zero index from the encoder. This process is comparable to MechanicalZeroAndIndexHomeSearch, but uses the minus end-of-run limit signal as hardware home switch and moves in the positive direction until the Index is reached. Otherwise, it will reach the positive limit or a timeout will occur. The positioner homes to a position that is different from the MechanicalZeroAndIndexHomeSearch location.
6. **MinusEndOfRunHomeSearch** uses the positioner's minus end-of-run limit for homing. Note that the emergency stop at the negative limit is disabled during homing.
7. **PlusEndOfRunHomeSearch** uses the positioner's plus end-of-run limit for homing and the emergency stop at the positive limit is disabled during homing.

The home search process is set up in the stages.ini file. When using the XPS controller with Newport ESP-compatible stages, this setting is done automatically with the configuration of the system. The home search velocity, acceleration and time-out are also set up in the stages.ini file.

Each motion group can either be homed "together" or "sequentially", meaning all positioners belonging to that group home at the same time in parallel or all the positioners home one after the other, respectively. This option is also set up in the system.ini file or during configuration.

A Home search can be executed with all motion groups and any motion group MUST be homed before any further motion can be executed. To home a motion group that is in a "ready" state, that motion group must first be "killed" and then "re-initialized".

Example

This is the sequence of functions that initialize and home a motion group.

GroupInitialize (MyGroup)

GroupHomeSearch (MyGroup)

…

GroupKill (MyGroup)

3.3 Referencing State

The predefined home search processes described in the previous section might not be compatible with all motion devices or might not be always executable. For instance, if there is a risk of collision during a standard home search process. In other situations, a home search process might not be desirable. For example, to ensure that the stages have not moved, the current positions are stored into memory. In this case, it is sufficient to reinitialize the system by setting the position counters to the stored position values.

For these special situations, the XPS controller's referencing state as in alternative to the predefined home search processes.

NOTE

The Referencing state should be only used by experienced users. Incorrect use could cause equipment damage.

The Referencing state is a parallel state to the homing state, see the state diagram on page 16, Figure 6. To enter the referencing state, send the function **GroupReferencingStart(GroupName)** while the group is in the NOT REFERENCED state.

In the Referencing state, the function **GroupReferencingActionExecute(PositionerName, Action, Sensor, Parameter)** will perform certain actions like moves, position latches of reference signal transitions, or position resets.

The function **PositionerSGammaParametersSet(PositionerName)** can be used to change the velocity, acceleration and jerk time parameters.

To leave the referencing state, send the function **GroupReferencingStop(GroupName)**. The Group will then be in the HOMED state, state number 11.

The syntax and function of the function **GroupReferencingActionExecute(PositionerName, Action, Sensor, Parameter)** will be discussed in detail. With this function, there are four parameters to specify:

- PositionerName is the name of the positioner on which this function is executed.
- Action is the type of action that is executed. There are eight actions that can be distinguished into three categories: Moves that stop on a sensor event, moves of certain displacement, and position counter reset categories.
- Sensor is the sensor used for those actions that stop on a sensor event. It can be MechanicalZero, MinusEndOfRun, or None.
- Parameter is either a position or velocity value and provides further input to the function.

The following table summarizes all possible configurations:

Action	Sensor			Parameter	
	MechanicalZero	MinusEndOfRun	None	Position	Velocity
LatchOnLowToHighTransition	■	■			■
LatchOnHighToLowTransition	■	■			■
LatchOnIndex			■		■
LatchOnIndexAfterSensorHighToLowTransition	■	■			■
SetPosition			■	■	
SetPositionToHomePreset			■		
MoveToPreviouslyLatchedPosition			■		■
MoveRelative			■	■	

3.3.1 Move on Sensor Eevents

The “move on sensor events” starts a motion at a defined velocity, latches the position when a state transition of a certain sensor is detected, then stops the motion. There are four possible actions under this category:

- LatchOnLowToHighTransition
- LatchOnHighToLowTransition
- LatchOnIndex
- LatchOnIndexAfterSensorHighToLow

With **LatchOnLowToHighTransition** and **LatchOnHighToLowTransition**, latching happens when the right transition on the defined sensor occurs. The sensor can be latched to either **MechanicalZero**, **MinusEndOfRun** and **PositiveEndOfRun** when supported by the hardware, refer to section 3.2: “Home Search” to know which hardware supports the function. With **LatchOnIndex** and **LatchOnIndexAfterSensorHighToLow**, latching happens on the index signal. With **LatchOnIndexAfterSensorHighToLow,** latching happens on the first index after a high to low transition at the defined sensor (**MechanicalZero** or **MinusEndOfRun**). Because of the dedicated hardware circuits used for the position latch, there is essentially no latency between sensor transition detection and position acquisition.

In all cases, motion stops after the latch. However, this means that the stopped position doesn’t rest on the sensor transition, but at some short distance from it. To move exactly to the position of the sensor transition, use the action **MoveToPreviouslyLatchedPosition**.

The latch does not change the current position value. In order to set the current position value, use the action **SetPosition** or **SetPositionToHomePreset**, for instance, after a **MoveToPreviouslyLatchedPosition**.

In the Referencing state, the limit switch safeties (emergency stop) are still enabled until the **MinusEndOfRun** sensor is specified with a **GroupReferencingActionExecute**() function. When specified, the limit switch safeties are disabled and will only be re-enabled with the function **GroupReferencingStop**().

The Parameter has a sign, if it is assigned as velocity (floating point). This means that the direction of motion is dictated by the sign of the velocity parameter.

3.3.2 Moves of Certain Displacements

These two move commands which don't use the same parameters, are explained below.

- MoveRelative

  The action **MoveRelative** commands a relative move of a positioner similar to the function **GroupMoveRelative**. However, the function **GroupMoveRelative** is not available in the Referencing state. The relative move is specified by a positive or negative displacement. The move is done with the SGamma profiler. The speed and acceleration are the default values, or the last value defined by either a move on sensor event, a **MoveToPreviouslyLatchedPosition**, or a **PositionerSGammaParametersSet**.

- MoveToPreviouslyLatchedPosition

  This action moves the positioner to the last latched position, see section 3.3.1: "Move on sensor events" for details. It verifies there was a position latched since this last **GroupReferencingStart** call. This is important because an old latched position can still be in memory from a previous home search or referencing. And moving to this previous latched position could have unexpected results. The move is done with the SGamma profiler. The speed is specified by a parameter. The acceleration is the default value, or the last value defined by a **PositionerSGammaParametersSet**.

3.3.3 Position Counter Resets

"Position counter resets" sets the current position to a certain value. There are two options: **SetPosition** and **SetPositionToHomePreset**. The main use of these actions is when the positioner is at a well defined reference position after a **MoveToPreviouslyLatchedPosition** action.

Another use of this action is for a "soft" system start by Referencing a group to a known set position, without executing a home search process, for example. In this case, a suggested sequence of functions follows:

GroupReferencingStart(GroupName)

GroupReferencingActionExecute(PositionerName, "SetPosition", "None", KnownCurrentPosition)

GroupReferencingStop(GroupName)

SetPosition sets the current position to a value defined by a parameter. **SetPositionToHomePreset** sets the current position to the **HomePreset** value stored in the stages.ini configuration file. It is equivalent to a **SetPosition** of the same positioner to the **HomePreset** value.

It is important that all positioners of a motion group are referenced to a position using the **SetPosition** or **SetPositionToHomePreset** before leaving the Referencing state (see example on page 94).

3.3.4 State Diagram

The Referencing state is a parallel state to the homing state. It is between the NotReferenced state and the Ready state. Please see the state diagram below:

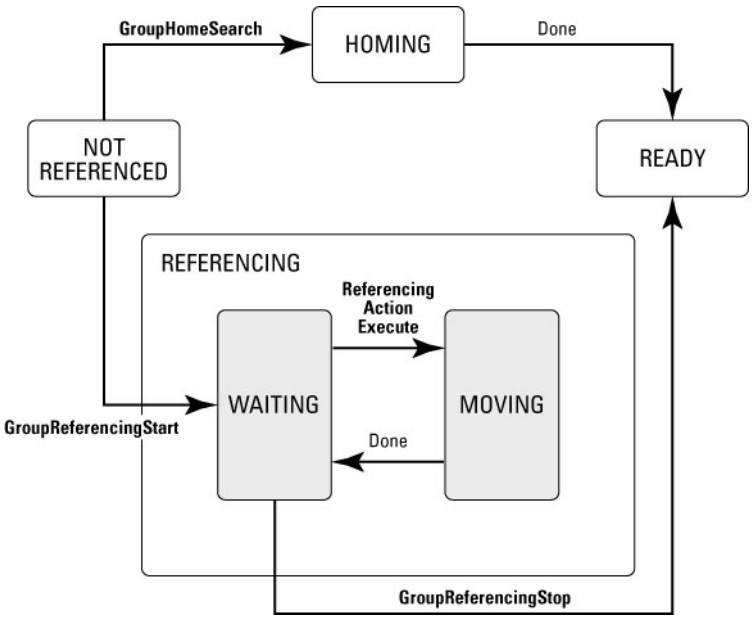


Figure 6: State diagram.

3.3.5 Example: MechanicalZeroAndIndexHomeSearch

The following sequence of functions has the same effect as the MechanicalZeroAndIndexHomeSearch:

```
GroupReferencingStart(GroupName)
PositionerHardwareStatusGet (PositionerName, &status)
if ((status & 4) == 0) { // 4 is the Mechanical zero mask on the
hardware status GroupReferencingActionExecute(PositionerName,
“LatchOnLowToHighTransition”, “MechanicalZero, -10) }
GroupReferencingActionExecute(PositionerName,
“LatchOnHighToLowTransition”, “MechanicalZero”, 10)
GroupReferencingActionExecute(PositionerName,
“LatchOnLowToHighTransition”, “MechanicalZero”, -5)
GroupReferencingActionExecute(PositionerName,
“LatchOnIndexAfterSensorHighToLow”, “MechanicalZero”, 5)
GroupReferencingActionExecute(PositionerName,
“MoveToPreviouslyLatchedPosition”, “None”, 5)
GroupReferencingActionExecute(PositionerName,
“SetPositionToHomepreset”, “None”, 0)
GroupReferencingStop(GroupName)
```

3.4 Move

A move is a point-to-point motion. On execution of a move command, the motion device moves from a current position to a desired destination (absolute move) or by a defined increment (relative move). During motion, the controller is monitoring the feedback of the positioner and is updating the output based upon the following error. The XPS controller’s position servo is updated at servo cycle rate. These defaults values

provide highly accurate closed loop positioning. Between the profiler and the corrector, there is a time-based linear interpolation to accommodate the different frequencies.

There are two types of moves that can be commanded: an absolute move and a relative move. For an absolute move, the positioner will move relative to the HomePreset position as defined in the stages.ini file. In most cases the HomePreset is 0, which makes the home position equal to the zero position of the positioner. For a relative move, the positioner will move relative to the current TargetPosition. In relative moves, it is possible to make successive moves that are not equal to a multiple of an encoder step without accumulating errors.

Absolute and relative moves can be commanded to positioners and to motion groups. When commanding a move to a positioner, only the position parameter for that positioner must be provided. When commanding a move to a motion group, the appropriate number of position parameters must be provided with the move command. For instance for a move command to an XYZ group, 3 position parameters must be defined.

When commanding a move to a motion group, all positioners of that group will move synchronously. For any move, the controller will always determine the shortest time within the positioner's parameters setup. All positioners will start and stop their motion at the same time. This type of motion is also known as linear interpolation.

The functions for absolute and relative motions are **GroupMoveAbsolute**() and **GroupMoveRelative**() respectively.

Example

A motion system consisting of one XY group called ScanTable and one SingleAxis group called FocusStage. ScanTable has two positioners, called ScanAxis and StepAxis.

…

GroupHomeSearch (ScanTable)

GroupHomeSearch (FocusStage)

After homing is completed…

GroupPositionCurrentGet (ScanTable, Pos1, Pos2)

… will return 0 to Pos1 and 0 to Pos2, assuming PresetHome = 0.

GroupPositionCurrentGet (FocusStage, Pos3)

Will return 0 to Pos3, assuming HomePreset = 0.

GroupMoveAbsolute (ScanTable, 100, 50)

GroupMoveAbsolute (ScanTable.StepAxis, -20)

The second move is only for one positioner of that group and can be only executed after the first move is completed. After all moves are completed…

GroupPositionCurrentGet (ScanTable, Pos1, Pos2)

… will return 100 to Pos1 and -20 to Pos2.

GroupMoveRelative (FocusStage, 1)

GroupMoveRelative (FocusStage, 1)

The second move can be only executed after the first move is completed. After all moves are completed…

GroupPositionCurrentGet (FocusStage, Pos3)

… will return 2 to Pos3.

The velocity, acceleration and jerk time parameters of a move are defined by the function PositionerSGammaParametersSet() (see also section 3.1). When the controller receives new values for these parameters during the execution of a move, it will not take these new values into account on the current move, but only on the following moves. To

change the velocity or acceleration of a positioner during the motion, use the Jogging mode (see section 3.5).

A move can be stopped at any time with the function **GroupMoveAbort**() that accepts GroupNames and PositionerNames. It is important to note, however, that the function **GroupMoveAbort(PositionerNames)** is accepted when the motion was commanded to the positioner, and not to the group. In the previous example, the function **GroupMoveAbort(ScanTable.ScanAxis)** is rejected for a motion that has been launched with **GroupMoveRelative(ScanTable, 100, 50)**. To stop this motion, send the function **GroupMoveAbort(ScanTable)**.

With XPS firmware 1.5.0 and higher, the XPS controller supports also asynchronous moves of several positioners belonging to the same motion group. The individual motion, however, needs to be managed by separate threads (see also section 13.3: "Running Processes in Parallel" for details).

3.5 Motion Done

The XPS controller supports two methods that define when a motion is completed (MotionDone): the theoretical MotionDone and the VelocityAndPositionWindow MotionDone. The method used is set in the stages.ini file. In theory, MotionDone is completed as defined by the profiler. However, it does not take into account the settling of the positioner at the end of the move. So depending on the precision and stability requirements at the end of the move, the theoretical MotionDone might not always be the same as the physical end of the motion. The VelocityAndPositionWindow MotionDone allows a more precise definition by specifying the end of the move with a number of parameters that take the settling of the positioner into account. In the VelocityAndPositionWindow MotionDone, the motion is completed when:

> | PositionErrorMeanValue | < | MotionDonePositionThreshold | AND | VelocityMeanValue | < | MotionDoneVelocityThreshold | is verified during the MotionDoneCheckingTime period.

The different parameters have the following meaning:

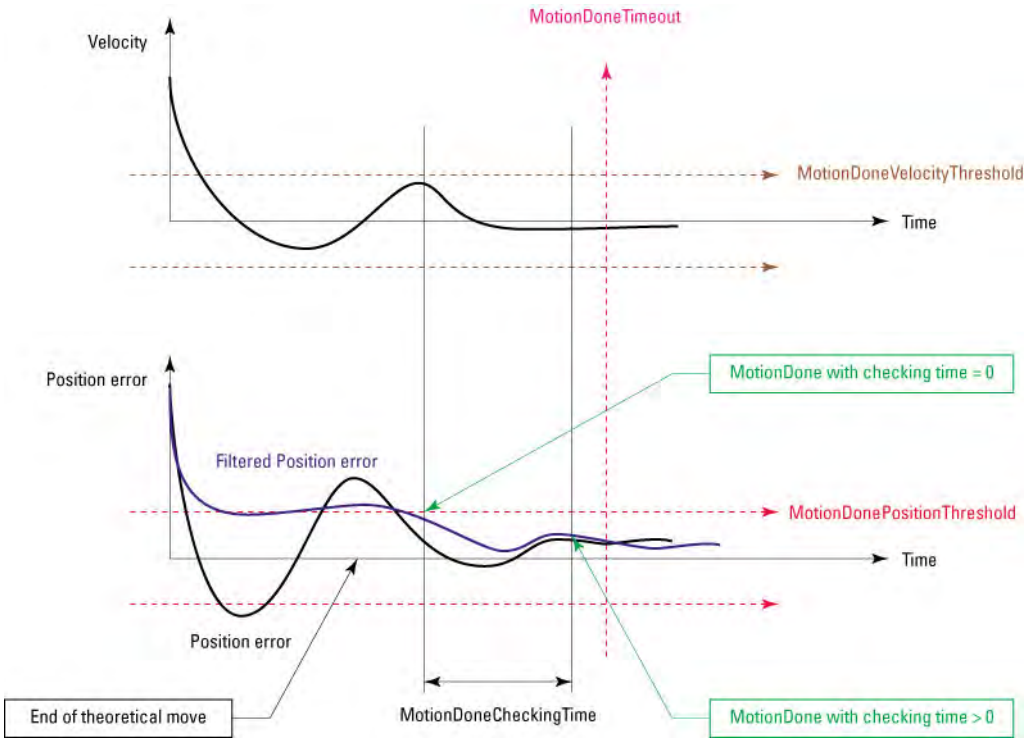


Figure 7: Motion done.

- **MotionDonePositionThreshold**: This parameter defines the position error window. The position error has to be within ± of this value for a period of MotionDoneCheckingTime to validate this condition.
- **MotionDoneVelocityThreshold**: This parameter defines the velocity window. The velocity at the end of the motion has to be within ± of this value for a period of MotionDoneCheckingTime to validate this condition.
- **MotionDoneCheckingTime**: This parameter defines the period during which the conditions for the MotionDonePositionThreshold and the MotionDoneVelocityThreshold must be true before setting the motion done.
- **MotionDoneMeanPeriod**: A sliding mean filter is used to attenuate the noise for the position and velocity parameters. The MotionDoneMeanPeriod defines the duration for calculating the sliding mean position and velocity. The mean position and velocity values are compared to the threshold values as defined above. This parameter is not illustrated on the graph.
- **MotionDoneTimeout**: This parameter defines the maximum time the controller will wait from the end of the theoretical move for the MotionDone condition, before sending a MotionDone time-out.

Important:

The XPS controller can only execute a new move on the same positioner or on the same motion group when the previous move is completed (MotionDone) and when the positioner or the motion group is again in the ready state.

The XPS controller allows triggering an action when the motion is completed (MotionDone) by using the event MotionEnd. For further details see chapter 7.0.

The functions **PositionerMotionDoneGet**() and **PositionerMotionDoneSet**() allow reading and modifying the parameters for the VelocityAndPositionWindow MotionDone. These parameters are only taken into account when the MotionDoneMode is set to VelocityAndPositionWindow in the stages.ini.

Example

Modifications of the MotionDoneMode can be made only manually in the stages.ini file. The stages.ini file is located in the config folder of the XPS controller, see section **User Interface Manual** for details. Stage parameters can also be modified from the website, in Administrator mode, STAGES menu, Modify submenu.

Make a copy of the stages.ini file to the PC. Open the file with any text editor and modify the MotionDoneMode parameter of the appropriate stage to VelocityAndPositionWindow, and set the following parameters:

```
;--- Motion done
MotionDoneMode = VelocityAndPositionWindow          ; instead of
Theoretical
MotionDonePositionThreshold = 4              ; units
MotionDoneVelocityThreshold = 100            ; units/s
MotionDoneCheckingTime = 0.1                 ; seconds
MotionDoneMeanPeriod = 0.001                 ; seconds
MotionDoneTimeout = 0.5                      ; seconds
```

Replace the current stages.ini file on the XPS controller with this modified version (make a copy of the old .ini file first). Reboot the controller. To apply any changes to the stages.ini or system.ini, the controller has to reboot.

Use the following functions:

GroupInitialize(MyGroup)

GroupHomeSearch(MyGroup)

PositionerMotionDoneGet(MyGroup.MyPositioner)

This function returns the parameters for the VelocityAndPositionWindow Motion done previously set in the stages.ini file, so 4, 100, 0.1, 0.001 and 0.5.

PositionerMotionDoneSet(MyGroup.MyPositioner, PositionThresholdNewValue, VelocityThresholdNewValue, CheckingTimeNewValue, MeanPeriodNewValue, TimeoutNewValue)

This function replaces the parameters with the newly entered values. If this function is not executed, the default setting from the .ini file is used.

3.6 JOG

Jog is an indeterminate motion defined by velocity and acceleration. Unlike a **GroupMoveAbsolute**() or a **GroupMoveRelative**(), the end of the motion is not defined by a target position. It can be best described by a "go"-command with a definition how fast, but not how far.

In Jog mode, the speed and acceleration of a motion group can be changed on-the-fly to accommodate varying situations. This is not possible with a **GroupMoveAbsolute**() or a **GroupMoveRelative**() which are defined moves. Practical examples for Jog are with tracking systems or coordinate transformations where the speed or acceleration of the jogging group is modified depending on the position or speed of the other motion groups or based on an analog input value.

The Jog mode can be enabled using the function **GroupJogModeEnable**() and is available to all motion groups. Once this mode is enabled, the motion parameters can be set using the command **GroupJogParameterSet**() which is applicable to positioners and to motion groups. To query the maximum jog velocity and acceleration values for a positioner, use **PositionerJogMaximumVelocityAndAccerationGet**(). To exit the Jog mode, first set the velocity to zero and then send the function **GroupJogModeDisable**().

Examples

For a single axis group:

GroupJogModeEnable (MySingleGroup)

Enables the Jog mode.

GroupJogParameterSet (MySingleGroup, 5, 20)

The single stage starts moving with a velocity of 5 units per second and an acceleration of 20 units per second^2.

GroupJogParameterSet (MySingleGroup, -5, 20)

The single stage starts moving in the reverse direction with the same velocity and same acceleration.

GroupJogParameterSet (MySingleGroup, 0, 20)

The single stage stops moving, its velocity being 0 units per second.

GroupJogModeDisable (MySingleGroup)

Disables the Jog mode.

For an XY group:

GroupJogModeEnable (MyXYGroup)

Enables the Jog mode.

GroupJogParameterSet (MyXYGroup, 5, 20, 10, 40)

The X axis and Y axis start moving with a velocity of 5 and 10 units per second and an acceleration of 20 and 40 units per second^2 respectively.

GroupJogParameterSet (MyXYGroup, 0, 20, 0, 40)

Both stages stop moving, their velocities being 0 units per second.

To apply new parameters to only one stage, use the following function:

GroupJogParameterSet (MyXYGroup.XPositioner, 5, 20)

Only the X axis starts moving with a velocity of 5 units per second and an acceleration of 20 units per second^2.

GroupJogParameterSet (MyXYGroup.XPositioner, 0, 20)

The X axis stage stops moving, its velocity being 0 units per second.

GroupJogModeDisable (MyXYGroup)

Disables the Jog mode.

In Jog mode, the profiler uses the CurrentPosition and the defined velocity and acceleration to calculate a new Setpoint position every 0.4 ms. These new Setpoint positions are then transferred to the corrector loop which runs every 0.1 ms. To accommodate the different frequencies between the profiler and the corrector, a linear interpolation between the new Setpoint and the previous Setpoint is done. Worst case, a new velocity and acceleration can be executed only every 0.4 ms. In Jog mode, the profiler uses a trapezoidal motion profile (see also section 3.1 for further details on motion profiles).

3.7 Master Slave

In master slave mode, any motion axis can be electronically geared to another motion axes, or a single master with multiple slaves. The gear ratio between the master and the slave is user defined. During motion, all axes compensations of the master and the slave are taken into account.

The slave must be a SingleAxis group. The master can be a positioner from any group. The Master slave relation is set by the function **SingleAxisSlaveParametersSet**().

The Master slave mode is enabled by the function **SingleAxisSlaveModeEnable**(). To enable the Master slave mode, the Slave group must be in the ready state. The Master group can be in the not-referenced or ready state.

Example 1

This example shows the sequence of functions used to set-up a master-slave relation between two axes that are not mechanically joined (meaning the two axis can move independently):

GroupInitialize (SlaveGroup)

GroupHomeSearch (SlaveGroup)

GroupInitialize (MasterGroup)

GroupHomeSearch (MasterGroup)

…

SingleAxisSlaveParametersSet (SlaveGroup, MasterGroup.Positioner, Ratio)

SingleAxisSlaveModeEnable (SlaveGroup)

GroupMoveRelative (MasterGroup.Positioner, Displacement)

…

SingleAxisSlaveModeDisable (SlaveGroup)

Example 2

This example shows the sequence of functions used to set-up a Master slave relation **between two axes that are mechanically joined**. Different from example 1, all motions, including the motion done during the home search routine, are performed synchronously.

Important: First, set the HomeSearchSequenceType of the Slave group's positioner to CurrentPositionAsHome in the stages.ini and reboot the XPS controller.

GroupInitialize (SlaveGroup)

GroupHomeSearch (SlaveGroup)

GroupInitialize (MasterGroup)

SingleAxisSlaveParametersSet (SlaveGroup, MasterGroup.Positioner, Ratio)

SingleAxisSlaveModeEnable (SlaveGroup)

GroupHomeSearch (MasterGroup)

…

GroupMoveRelative (MasterGroup.Positioner, Displacement)

NOTE

The slave positioners should have similar capabilities as the master positioner in terms of velocity and acceleration. Otherwise the full capabilities of the master or the slave positioners may not be utilized.

3.8 Analog Tracking

Analog tracking controls the position or velocity of a motion group via external analog inputs. Analog tracking is available with all motion groups. To enable this mode, first set the tracking parameters of the positioners belonging to that motion group. Then enable tracking while the motion group is homed (in ready state after homing). In analog tracking mode, the analog inputs are filtered by a first order low-pass filter. Its cut-off frequency is defined by the parameter "TrackingCutOffFrequency" given in the section "profiler" of the stage.ini parameter file.

To set or get the tracking parameters, use the following functions:

PositionerAnalogTrackingPositionParametersSet()

PositionerAnalogTrackingPositionParametersGet()

…

PositionerAnalogTrackingVelocityParametersSet()

PositionerAnalogTrackingVelocityParametersGet()

The functions PositionerAnalogTrackingPositionParametersSet() and PositionerAnalogTrackingVelocityParametersSet() define the maximum velocity and acceleration used during analog tracking.

3.8.1 Analog Position Tracking

The parameters that can be set for analog position tracking are the GPIO Name, scale and offset. The GPIO Name denotes which connector and pin number the analog signal will be input. The scale and the offset are used to calibrate the output position in the following way:

Position = InitialPosition + (AnalogValue - Offset) * Scale

Typical applications of analog position tracking are for beam stabilization, tracking systems, auto focusing sensors or alignment systems. When connecting a function

generator to the GPIO input, analog tracking provides an easy way to make cyclical or sinusoidal motion, for example.

Example

Following is an example that shows the sequence of functions used to setup Analog Position Tracking:

GroupInitialize (Group)

GroupHomeSearch (Group)

…

PositionerAnalogTrackingPositionParameterSet (Group.Positioner, GPIO4.ADC1, Offset, Scale, Velocity, Acceleration)

GroupAnalogTrackingModeEnable (Group, "Position")

…

GroupAnalogTrackingModeDisable (Group)

3.8.2 Analog Velocity Tracking

The parameters that can be set for analog velocity tracking are the GPIO Name, offset, scale, deadband threshold and order. The relationship among offset, scale, deadband and order is illustrated in Figure 8.

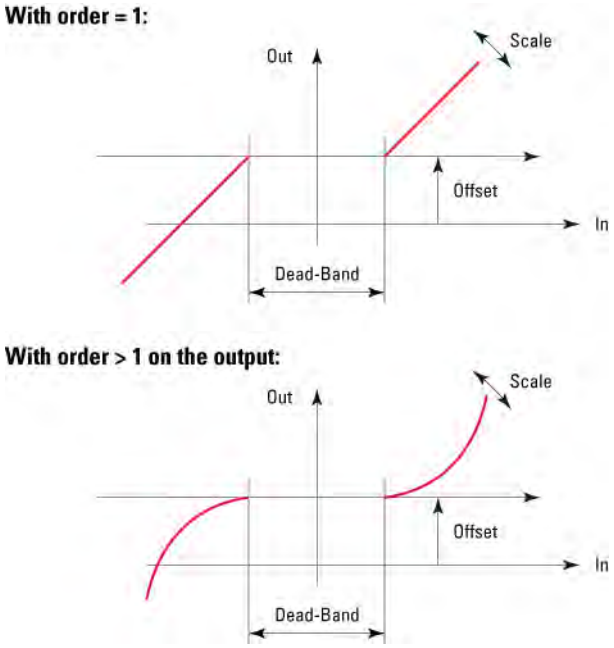


Figure 8: The relationship among Offset, Scale, Dead Band & Order.

The tracking velocity calculates as follows:

- AnalogInput is the voltage input at the GPIO
- AnalogGain refers to the AnalogGain setting of the analog input
- Offset, Order, DeadBandThreshold, and scale are defined with the function PositionerAnalogTrackingVelocityParametersSet
- MaxADCAmplitude, InputValue, OutputValue are internally-used parameters only

InputValue = AnalogInput - Offset

if (InputValue >= 0) **then**

InputValue = InputValue - DeadBandThreshold

if (InputValue < 0) **then** InputValue = 0

else

InputValue = InputValue + DeadBandThreshold

if (InputValue > 0) **then** InputValue = 0

OutputValue = (|InputValue|/ MaxADCAmplitude) * Order

Velocity = Sign(InputValue) * OutputValue * Scale * MaxADCAmplitude

In the dead band region there is no motion. If the order is set to 1, then the velocity is linear with respect to the input voltage.

If order is set greater than 1, then the velocity response is polynomial with respect to the input voltage. This makes the change in velocity more gradual and more sensitive in relation to the change in voltage.

A good example for using analog velocity tracking is for an analog joystick.

Example

Following is an example that shows the sequence of functions used to set-up Analog Velocity Tracking:

GroupInitialize (Group)

GroupHomeSearch (Group)

…

PositionerAnalogTrackingVelocityParameterSet (Group.Positioner, GPIO4.ADC1, Offset, Scale, DeadBandThreshold, Order, Velocity, Acceleration)

GroupAnalogTrackingModeEnable (Group, "Velocity")

…

GroupAnalogTrackingModeDisable (Group)

4.0 Trajectories

The XPS controller supports 4 different types of trajectories:

The Line-arc trajectory is a trajectory defined by a combination of straight and curved segments. It is available only for positioners in XY groups. The major benefit of a Line-arc trajectory is the ability to maintain constant speed (speed being the scalar of the trajectory velocity) throughout the entire path, excluding the acceleration and deceleration periods. The trajectory is user defined in a text file that is sent to the controller via FTP. Once defined, the user executes a function to begin the trajectory and the XPS automatically calculates and executes the motion, including precise monitoring of the speed and acceleration all along the trajectory. Simply executing the same trajectory more than once results in continuous path contouring. A dedicated function performs a precheck of the trajectory which returns the maximum and minimum travel requirements per positioner as well as the maximum possible trajectory speed and trajectory acceleration that is compatible with the different positioner parameters.

The spline trajectory executes a Catmull-Rom spline (which is a 3rd order polynomial curve) on an XYZ group. The main requirements of a spline are to hit all points (except for the first and the last point that are only needed to define the start and the end of the trajectory) and to maintain a constant speed throughout the entire path (except during the acceleration and deceleration period). The definition and execution of the spline trajectory is similar to the Line-arc trajectory with similar functions for trajectory pre-checking.

The PVT-mode is the most complex trajectory and is only available with MultipleAxes group. In a PVT trajectory, each trajectory element is defined by the displacement and end speed of each positioner plus the move time for the element. When all elements are defined, the controller calculates the cubic function trajectory that will pass through all defined positions at the defined times and velocities. PVT is a powerful tool for any kind of trajectory with varying speeds and for trajectories with rotation stages or other nonlinear motion devices.

The PT-mode is based on a simpler definition of PVT trajectory and is only available with MultipleAxes group. In a PT trajectory, each trajectory element is defined by the displacement and move time for the element. When all elements are defined, the controller calculates the cubic function trajectory that will pass through all defined positions at the defined times. The output velocity of each element is defined by the firmware to avoid speed oscillations when successive elements are set with the same speed (DX/DT= constant).

4.1 Line-Arc Trajectories

4.1.1 Trajectory Terminology

Trajectory: defined as a continuous multidimensional motion path. Line-arc trajectories are defined in a two-dimensional XY plane. These are used with XY groups. The main requirement of a Line-arc trajectory is to maintain a constant speed (speed being the scalar of the vector velocity) throughout the entire path (except during the acceleration and deceleration periods).

Trajectory element (segment): an element of a trajectory is defined by a simple geometric shape, in this case a line or an arc segment.

Trajectory velocity: the tangential linear velocity (speed) along the trajectory during its execution.

Trajectory acceleration: the tangential linear acceleration used to start and end a trajectory. Trajectory acceleration and trajectory deceleration are equal by default.

4.1.2 Trajectory Conventions

When defining and executing a Line-arc trajectory, a number of rules must be followed:

- The motion group must be an XY group.
- All trajectories must be stored in the controller's memory under ..\public\trajectories (one file for each trajectory). Once a trajectory is started, it executes in the background allowing other groups or positioners to work independently and simultaneously.
- Each trajectory must have a defined beginning and end. Endless (infinite) trajectories are not allowed. Although, N-times (N defined by user) non-stop execution of the same trajectory is allowed. As the trajectory is stored in a file, the trajectory's maximum size (maximum elements number) is unlimited for practical purposes.
- Two types of Line-arc trajectory elements (segments) are available: lines Line(X,Y) and arcs Arc(R,A) (Radius, SweepAngle). Any Line-arc trajectory is a set of consecutive line or arc segments. The line segments are true linear interpolations $y = A*x + B$, the arc segments are true arcs of circles $(x - x0)^2 + (y - y0)^2 = R^2$.
- A Line-arc trajectory forms a continuous path, so each segment's final position is equal to the next segment's starting position. However, as the segment's tangential angles around the connection point of any two consecutive segments may not be continuous, there might be velocity discontinuities from one segment to next. For reference, this discontinuity is categorized as R0, wherein the position is continuous, but velocity is not. An excessive velocity discontinuity at joints can damage the stages, so the trajectory definition process must take this into account.
- Each Line-arc trajectory element is defined relative to the trajectory starting point. Every trajectory starting point has the coordinates (0,0), which has no relation to the zero position of the positioners. All trajectories physically start from the current X and Y positions of the XY group.

4.1.3 Geometric Conventions

The coordinate system of a Line-arc trajectory is an XY orthogonal system.

The X-axis of this system correlates to the XPositioner and the Y-axis correlates to the YPositioner of the XY group as defined in the system.ini.

The origin of the XY coordinate system is in the lower left corner, with positive values up and to the right.

All angles are measured in degrees, presented as floating point numbers. Angle origin and signs follow the trigonometric convention: positive angles are measured counter-clockwise.

4.1.4 Defining Line-Arc Trajectory Elements

A Line-arc trajectory is defined by a number of line and arc elements. The trajectory elements are executed in the same order as defined in the trajectory data file.

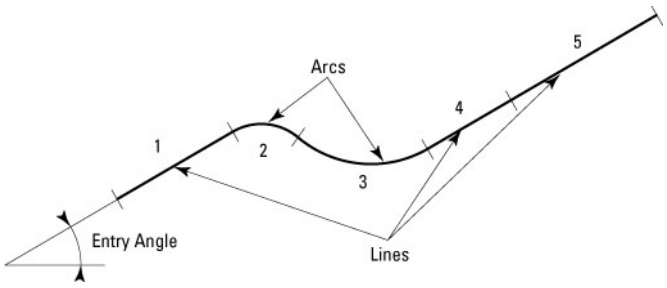


Figure 9: Line-arc trajectory example.

Figure 9 shows a trajectory example. Every trajectory must have a first element entry angle (called First Tangent) defined in the head of the trajectory data file. If the first element is a line, this parameter has no effect. If the first element is an arc, the entry

angle is the tangent to the first point of the arc. Each trajectory element is identified by a number, starting from 1. The references for synchronizing external events with the trajectory execution are the starting and ending points of these elements.

Line and arc elements can be sequenced in any order. An arc is automatically placed by the controller so that its entry angle corresponds to the exit angle of the preceding element to ensure the continuity of the trajectory. But with every line segment, the user must choose the (X,Y) end-point in that way that the angle discontinuity to the previous segment does not exceed the maximum allowed angular discontinuity. The angular discontinuity is measured in degrees and is defined in the head of the trajectory data file. In theory, a trajectory can be defined only by straight lines, if two adjacent line segments have an angular difference smaller than the allowed angle of discontinuity, as shown in the Figure 10.

Figure 10: Contouring with linear lines only.

In practice this is not recommended since each angle of discontinuity corresponds to an instantaneous velocity change on both axes, which produces large accelerations. This can result in a shock to the stages and an increase in the following error. The larger the angle of discontinuity, the larger the shock and following error will be. Special consideration must be given to both these effects when increasing the maximum discontinuity angle from its default value.

4.1.5 Define Lines

A line element is defined by specifying the (X_i, Y_i) ending point.

The succeeding element's starting point is always the end point of the previous segment (X_{i-1}, Y_{i-1}).

Note that all line element positions are defined relative to the trajectory's starting point (0, 0).

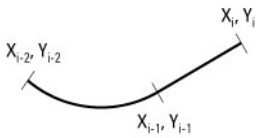


Figure 11: Line element to (X_i, Y_i) position coordinates.

As described before, when adding a new line element, the user must make sure that the discontinuity angle between the new segment and the previous one is not excessive.

4.1.6 Define Arcs

An arc is defined by specifying the radius R and the sweep angle A (Figure 12).

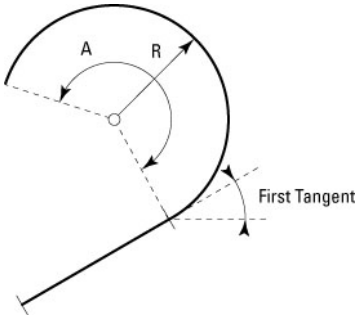


Figure 12: An arc defined with radius and angle.

Both radius and sweep angles are expressed in double precision floating point numbers. The sweep angle can range from 10^{-14} to 1.797×10^{308} allowing a definition of arcs from a fraction of a degree to practically an infinite number of overlapping circles.

4.1.7 Trajectory File Description

The Line-arc trajectory is defined in a file that has to be stored in the ..\public\trajectories folder of the XPS controller. This file must have the following structure:

The first line sets the “FirstTangent”:	Defines the tangent angle for the first point in case of an arc. This parameter has no effect if the first element is a line.
The second line sets the “DiscontinuityAngle”:	Defines the maximum allowed angle of discontinuity.
The third line must be empty for better readability.	
The following lines define the Line-arc trajectory:	Each line defines an element of the trajectory.

An element can be a “Line” or an “Arc”:

Line: Define X and Y positions to build a linear segment Line = X, Y.

Arc: Define radius and sweep angle to build an arc of circle Arc = R, A.

4.1.8 Trajectory File Examples

The following is an example of a trajectory file that represents a rectangle with rounded corners and with the end point equal to the starting point:

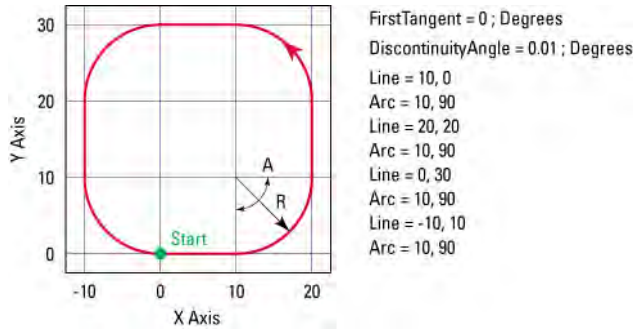


Figure 13: Graphical display of the first Line-arc trajectory data file example.

The following is an example of a trajectory file that represents a rectangle with rounded corners and with the end point equal to the starting point:

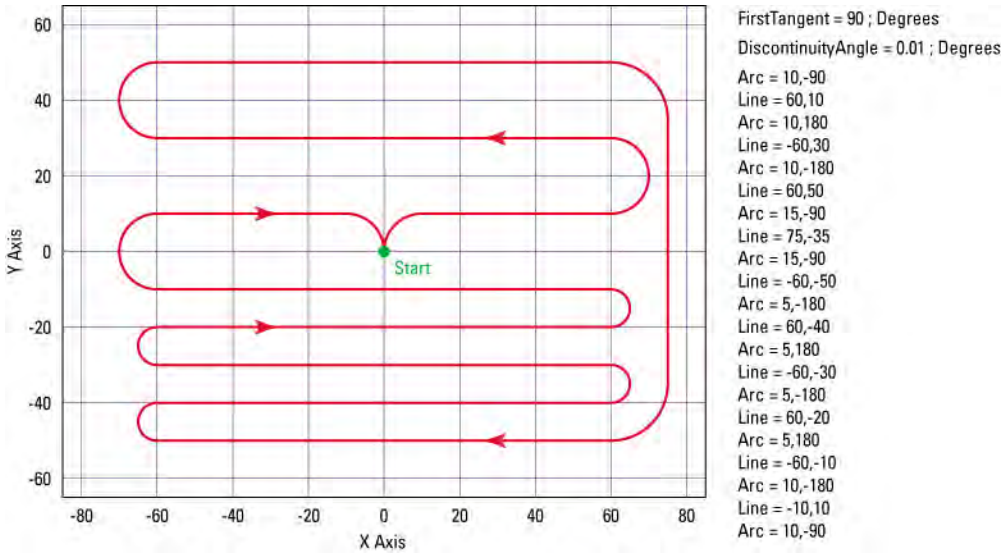


Figure 14: Graphical display of the second Line-arc trajectory data file example.

4.1.9 Trajectory Verification and Execution

There are four functions to verify or execute a Line-arc trajectory:

- **XYLineArcVerification(): ** Verifies a Line-arc trajectory data file.
- **XYLineArcVerificationResultGet(): ** Returns the last trajectory verification results, actuator by actuator. This function works only after an XYLineArcVerification().
- **XYLineArcExecution(): ** Executes a trajectory.
- **XYLineArcParametersGet(): ** Returns the trajectory's current execution parameters. This function works only while executing the trajectory.

The **function XYLineArcVerification**() can be executed at any time and is independent from trajectory execution. This function performs the following:

- Checks the trajectory file for data and syntax coherence.
- Calculates the trajectory limits, which are: the required travel per positioner, the maximum possible trajectory velocity and the maximum possible trajectory acceleration. This function defines the parameters for trajectory execution.
- If all is OK, it returns an "OK" (0). Otherwise, it returns a corresponding error.

The function **XYLineArcVerificationResultGet**() can be executed only after an XYLineArcVerification() and returns the following:

- Travel requirement in positive and negative direction for each positioner.
- The maximum possible trajectory velocity (speed) that is compatible with all positioner's velocity parameters. It returns a value for the trajectory velocity, that when applied, at least one of the positioners will reach its maximum allowed speed at least once along the trajectory. So the returned value varies between Min $\{V_{max\_actuator}\}$ and $velocity = \sqrt{\sum PositionerMaximumVelocity^2}$. However, this value does not take into account the positioners's acceleration, which can also limit the trajectory velocity. For example, the case of a Line-arc trajectory containing arc segments with a small radius.
- The maximum possible trajectory acceleration that is compatible with all positioners' parameters. This means that one of the positioners will reach its maximum allowed acceleration during the trajectory execution.

The XYLineArcVerificationResultGet() function returns the trajectory execution limits that have previously been calculated by the XYLineArcVerification function. Note about this function's result: Only the returned travel requirements are specific for each

positioner. The returned velocity/acceleration values are the same for all positioners, because they represent the trajectory's velocity/acceleration.

To execute a Line-arc trajectory, send the function XYLineArcExecution() with the parameters for the trajectory velocity, and the trajectory acceleration that is used during the start and end of the trajectory. The motion profile for Line-arc trajectories is trapezoidal. The function XYLineArcExecution() does not verify the trajectory coherence or geometric conditions (exceeding any positioners, min. or max. travel, speed or acceleration) before execution, so users must pay attention when executing a trajectory and verify the trajectory relative to the maximum possible values or possible interference. In case of an error during execution, because of bad data or because of a following error (for example if the trajectory acceleration or speed was set too high) the motion group will make an emergency stop and will enter the disabled state. The parameters for trajectory velocity and trajectory acceleration can also be set to zero. In this case the controller uses executable default values which are Min{All $V_{max\_actuator}$} for trajectory velocity and Min{All $A_{max\_actuator}$} for trajectory acceleration.

A trajectory can be executed many times (up to 2^{31} times) by specifying the ExecutionNumber parameter with the XYLineArcExecution function. In this case, the second run of the trajectory is simply appended to the end of the first run, while the end position of the first run is taken as a new start position (referenced to zero) of the second run. The trajectory endpoint does not need to be the same as the start point. The total trajectory is executed without stopping between the different runs.

Finally, the function **XYLineArcParametersGet**() returns the trajectory execution status with trajectory name, trajectory velocity, trajectory acceleration and current executed trajectory element. This function returns an error if the trajectory is not executing.

4.1.10 Examples of the Use of the Functions

XYLineArcVerification (XYGroup, Linearc1.trj)

This function returns a 0 if the trajectory is executable.

XYLineArcVerificationResultGet (XYGroup.XPositioner, *Name, *NegTravel, *PosTravel, *MaxSpeed, *MaxAcceleration)

This function returns the name of the trajectory checked with the last sent function XYLineArcVerification to that motion group (Linearc1.trj), the negative or left travel required for the XYGroup.XPositioner, the positive or right travel required for the XYGroup.XPositioner, the maximum trajectory velocity and the maximum trajectory acceleration.

XYLineArcExecution (XYGroup, Linearc1.trj, 10, 100, 2)

Executes the trajectory Linearc1.trj with a trajectory velocity of 10 units/s and a trajectory acceleration of 100 units/s^2 two (2) times.

XYLineArcParametersGet (XYGroup, *FileName, *TrajectoryVelocity, *TrajectoryAcceleration, *ElementNumber)

Returns the name of the trajectory in execution (Linearc1.trj), the trajectory velocity (10), the trajectory acceleration (100) and the number of the current executed trajectory element.

4.2 Splines

4.2.1 Trajectory Terminology

Trajectory: Continuous multidimensional motion path. Spline trajectories are defined in a three-dimensional XYZ space. They are available with XYZ groups only. The major benefit provided by a spline trajectory is to hit all points (except for the first and the last point that are needed to define the start and the end) and to maintain an almost constant speed (speed being the scalar of the vector velocity) throughout the entire path (except during the acceleration and deceleration periods). Please note that the trajectory speed can vary in some areas depending on the distribution of the reference points. This is related to the spline algorithm used.

Trajectory element (segment): An element of a spline trajectory is defined by a 3rd order polynomial curve joining two consecutive control points.

Trajectory velocity: The tangential linear velocity (speed) along the trajectory during its execution.

Trajectory acceleration: The tangential linear acceleration used to start and end a trajectory. Trajectory acceleration and trajectory deceleration are always equal and by default.

4.2.2 Trajectory Conventions

When defining and executing a spline trajectory, a number of rules must be followed:

- The motion group must be an XYZ group.
- All trajectories must be stored in the controller's memory under ..\public\trajectories (one file for each trajectory). Once a trajectory is started, it executes in the background allowing other groups or positioners to work independently and simultaneously.
- Each trajectory must have a defined beginning and end. Endless (infinite) trajectories are not allowed. Although, N-times (N defined by user) non-stop execution of a trajectory is allowed. As the trajectory is stored in a file, the trajectory's maximum size (maximum elements number) is unlimited for practical purposes.
- Spline trajectory elements (segments) are 3rd order polynomial curve segments $S_i(u)$, joining the positions P_{i-1} (X_{i-1}, Y_{i-1}, Z_{i-1}) and P_i (X_i, Y_i, Z_i). Here "u" is the normalized time parameter that varies from 0 (corresponding to P_{i-1}) to 1 (corresponding to P_i).
- Spline trajectories form a continuous path (each segment's output position is equal to the next segment's input position), and the segment tangential angles at the connection point of any two consecutive segments are continuous, including its derivative. For reference, this discontinuity is categorized as R^1, wherein position and velocity are continuous, but not acceleration.

4.2.3 Geometric Conventions

The Spline trajectory's coordinate system is an XYZ orthogonal system.

The X-axis of this system correlates to the XPositioner, the Y-axis to the YPositioner, and the Z-axis to the ZPositioner of the XYZ group as defined in the stages.ini.

The origin of the XYZ coordinate system is in the lower left corner, with positive values up (Z), to the right (X) and forward (Y).

All angles are measured in degrees, presented as floating point numbers. Angle origin and sign follow the trigonometric convention: positive angles are measured counter-clockwise.

4.2.4 Catmull-Rom Interpolating Splines

To trace a smooth curve that links different predefined trajectory points, the intermediate points must be calculated following a mathematical model. For the sake of simplicity, in most cases this is done by a polynomial curve (polynomial interpolation). For motion systems, the resulting curve should hit all predefined points. This is called precise interpolation in contrast to approximate interpolation (like Bezier splines), where the predefined points act only as control points. Within this class of precise interpolation are:

- Global polynomial interpolation: One polynomial represents the whole trajectory. Examples are Lagrange polynomials or Newton polynomials.
- Local polynomial interpolation: Each segment that links two consecutive trajectory points has its own polynomial. The resulting curve is obtained by segment polynomial concatenation. To limit oscillations inside segments, the polynomial order is generally limited to 3 or less. This is called spline interpolation. If the polynomial order is equal to 3, it is called cubic spline interpolation.

The interpolation methods are also classified by the continuity criterion C^k. An interpolating curve has the continuity C^k if it and its derivatives up to k-degrees are continuous in all its points. The interpolating spline curves generally have C^1 or C^2 continuity.

Catmull-Rom splines are a family of **local cubic interpolating splines** where the tangent at each point p_i is calculated based on the previous p_{i-1} and the next point p_{i+1} on the spline. In case of the spline curve tension $\tau = 1/2$ (normal case), the **Catmull-Rom** spline is described by the following equation:

$$S(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \cdot \frac{1}{2} \cdot \begin{pmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{pmatrix}$$

Here, p_i are the coordinates of the predefined trajectory point in x, y and z (p_{xi}, p_{yi}, p_{zi}). "u" is the normalized interpolating parameter, varying from 0 (starting at p_i) to 1 (ending at p_{i+1}).

Catmull-Rom splines have a C^1 continuity (continuity up to the first derivative), local control and interpolation. **Catmull-Rom** splines have the advantage of simple calculation without matrix inversion for on-line calculations, which is a great advantage for splines with a large number of trajectory points. For this reason, the XPS controller uses the **Catmull-Rom** spline interpolation.

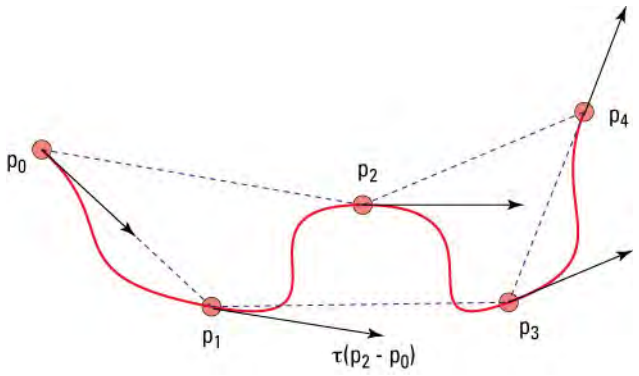


Figure 15: A Catmull-Rom spline.

4.2.5 Trajectory Elements Arc Length Calculation

Spline contouring at constant speed requires an accurate calculation of the segment's arc length. The segment's arc length can be expressed as follows:

$$L(u_0,u_1) = \int_{u_0}^{u_1} \sqrt{\left(\frac{d}{du}Sx(u)\right)^2 + \left(\frac{d}{du}Sy(u)\right)^2 + \left(\frac{d}{du}Sz(u)\right)^2}\,du$$

Here, $u_0 = 0$ is the segment starting point and $u_1 = 1$ is the segment ending point. Sx, Sy, Sz are x-, y-, and z-components of the segment function.

This integral can only be numerically calculated, which is done by the XPS controller using the Romberg numerical integration algorithm. This guarantees that the arc length is calculated with an error less than 10^{-7} units.

4.2.6 Trajectory File Description

The spline trajectory is described in a file in the \Admin\Public\Trajectories folder of the XPS controller. Each line of this file represents one point of the spline trajectory except for the first and the last lines that are needed only to define the start and the end of the trajectory. Two consecutive points form a trajectory segment.

The format of a line in a file is:

X-POSITION, Y-POSITION, Z-POSITION

The separator between the X-, Y-, and Z-Position is a comma.

As mentioned before, the first and last lines of the file are needed only for the interpolation of the first and the last spline segments. These define the angle the trajectory starts and ends, but the motion system will not hit these points. So the trajectory's first "real" point (starting point) is the one defined by the second line and the trajectory's real "last" point (end point) is the one defined by the second to the last line.

The position values in the data file are relative to the physical position of the motion group at the start of the trajectory. If the position in the second line of the file (starting point) is not equal to zero (0, 0, 0), the real trajectory positions (those that the motion group will hit) are shifted further by this value.

Example

The spline trajectory file has the following format:

$$\begin{matrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ \dots & \dots & \dots \end{matrix}$$

At the moment the trajectory is executed, the motion group is at the position X_C, Y_C, Z_C. So the real matrix in absolute coordinates of the motion group is:

$$\begin{matrix} x_{c+x_0-x_1} & y_{c+y_0-y_1} & z_{c+z_0-z_1} \\ x_c & y_c & z_c \\ x_{c+x_2-x_1} & y_{c+y_2-y_1} & z_{c+z_2-z_1} \\ x_{c+x_3-x_1} & y_{c+y_3-y_1} & z_{c+z_3-z_1} \\ x_{c+x_4-x_1} & y_{c+y_4-y_1} & z_{c+z_4-z_1} \\ \dots & \dots & \dots \end{matrix}$$

4.2.7 Trajectory File Example

This trajectory example represents a spiral starting from (0, 20, 0) and ending at (0, -20, 24). As described before, the trajectory's first (-5, 19.365, -1) and last (5, -19.365, 25) points are only needed to define the start and end conditions of the trajectory. Because

the second line (0, 20, 0) is not equal to zero (0, 0, 0), all points that the motion group will hit during the execution of the trajectory are reduced by this value from the physical starting position of the motion group.

The original data file is (except for the tabs that are only added for better readability):

-5,	19.365,	-1	-15,	13.229,	13
0,	20,	0	-10,	17.321,	14
5,	19.365,	1	-5,	19.365,	15
10,	17.321,	2	0,	20,	16
15,	13.229,	3	5,	19.365,	17
20,	0,	4	10,	17.321,	18
15,	-13.229,	5	15,	13.229,	19
10,	-17.321,	6	20,	0,	20
5,	-19.365,	7	15,	-13.229,	21
0,	-20,	8	10,	-17.321,	22
-5,	-19.365,	9	5,	-19.365,	23
-10,	-17.321,	10	0,	-20,	24
-15,	-13.229,	11	5,	-19.365,	25
-20,	0,	12			

With this data file, the real trajectory points relative to the physical start position of the motion group are (first and last lines are eliminated because the motion group will not hit these points and the values from the second column are reduced by 20 as the first line was (0, 20, 0)):

0,	0,	0	-15,	-6.771,	13
5,	-0.635,	1	-10,	-2.679,	14
10,	-2.679,	2	-5,	-0.635,	15
15,	-6.771,	3	0,	0,	16
20,	-20,	4	5,	-0.635,	17
15,	-33.229,	5	10,	-2.679,	18
10,	-37.321,	6	15,	-6.771,	19
5,	-39.365,	7	20,	-20,	20
0,	-40,	8	15,	-33.229,	21
-5,	-39.365,	9	10,	-37.321,	22
-10,	-37.321,	10	5,	-39.365,	23
-15,	-33.229,	11	0,	-40,	24
-20,	-20,	12			

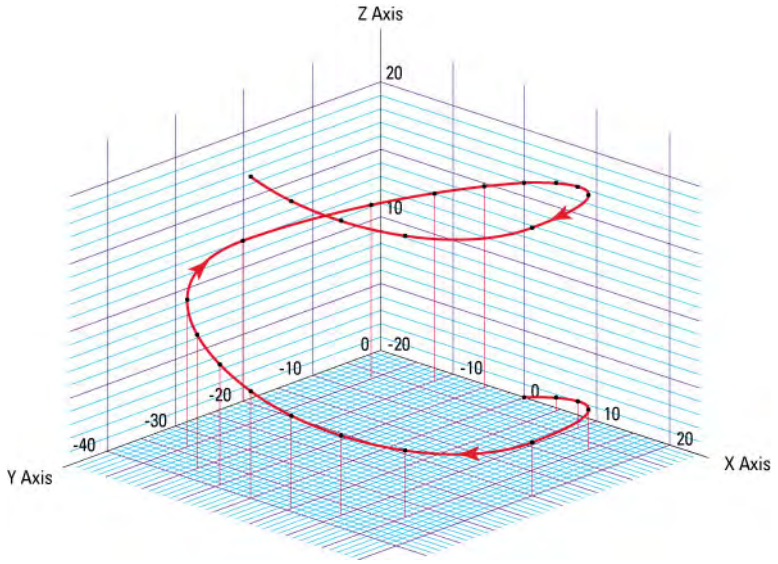


Figure 16: Executing the above normalized trajectory data file with the Catmull-Rom spline algorithm.

4.2.8 Spline Trajectory Verification and Execution

Here are four functions to verify or execute a spline trajectory:

- **XYZSplineVerification**(): Verifies a spline trajectory data file.
- **XYZSplineVerificationResultGet**(): Returns the last trajectory verification results, actuator by actuator. This function works only after an XYZSplineVerification().
- **XYZSplineExecution**(): Executes a trajectory.
- **XYZSplineParametersGet**(): Returns the trajectory current execution parameters. This function works only while executing of the trajectory.

The function **XYZSplineVerification**() can be executed at any moment and is independent from the trajectory execution. This function performs the following:

- Checks the trajectory file for data and syntax coherence.
- Calculates the trajectory limits, which are the required travel per positioner, the maximum possible trajectory velocity and the maximum possible trajectory acceleration. This function defines the parameters for trajectory execution.
- If all is OK, it returns an "OK" (0). Otherwise, it returns a corresponding error.

The function **XYZSplineVerificationResultGet**() can be executed only after an XYZSplineVerification() and returns the following:

- Travel requirement in the positive and negative directions for each positioner.
- The maximum possible trajectory velocity (speed) that is compatible with all positioners' velocity parameters. It returns a value for the trajectory velocity, that when applied, at least one of the positioners will reach its maximum allowed speed at least once along the trajectory. So the returned value varies between Min{Vmax_actuator} and $volocity = \sqrt{\sum PositionerMaximumVelocity^2}$. However, this value does not take into account that the positioners' acceleration can limit the trajectory velocity. This is the case with splines that contain sharp curved segments.
- The maximum trajectory acceleration that is compatible with all positioner parameters. At this trajectory acceleration, one of the positioners will reach its maximum allowed acceleration during trajectory execution.

The function **XYZSplineVerificationResultGet**() returns the trajectory execution limits that have previously been calculated by the XYZSplineVerification function. Note on this function's response: Only the returned travel requirements are specific for each positioner, the returned velocity/acceleration values are the same for all positioners, because they represent the trajectory's velocity/acceleration.

To execute a spline trajectory, send the function **XYZSplineExecution**() with the parameters for the trajectory velocity and the trajectory acceleration (the trajectory acceleration that is used during the start and the end of the trajectory). The motion profile for spline trajectories is trapezoidal. The function XYZSplineExecution() does not verify the trajectory's coherence or geometric conditions (exceeding any positioner's min. or max. travel, speed or acceleration) before execution, so users must pay attention when executing a trajectory without verifying the trajectory the maximum possible values. In case of an error during execution, because of bad data or because of a following error (for example the trajectory acceleration or speed was set too high) the motion group will make an emergency stop and will go to the disabled state. The parameters for trajectory velocity and trajectory acceleration can also be set to zero. In this case the controller uses executable default values which are the Min{All $V_{max\_actuator}$} for trajectory velocity and Min{All $A_{max\_actuator}$} for trajectory acceleration.

Finally, the function **XYZSplineParametersGet**() returns the trajectory execution status with trajectory name, trajectory velocity, trajectory acceleration and current executed trajectory element. This function returns an error if the trajectory is not executing.

4.2.9 Examples

XYZSplineVerification (XYZGroup, Spline1.trj)

This function returns a 0 if the trajectory is executable.

XYZSplineVerificationResultGet (XYZGroup.XPositioner, *Name, *NegTravel, *PosTravel, *MaxSpeed, *MaxAcceleration)

This function returns the name of the trajectory checked with the last sent function XYZSplineVerification to that motion group (Spline1.trj), the negative travel required for the XYZGroup.XPositioner, the positive travel required for the XYZGroup.XPositioner, the maximum trajectory velocity and the maximum trajectory acceleration.

XYZSplineExecution (XYZGroup, Spline1.trj, 10, 100)

Executes the trajectory Spline1.trj with a trajectory velocity of 10 units/s and a trajectory acceleration of 100 units/s^2.

XYZSplineParametersGet (XYZGroup, *FileName, *TrajectoryVelocity, *TrajectoryAcceleration, *ElementNumber)

Returns the name of the trajectory being executed (Spline1.trj), the trajectory velocity (10), the trajectory acceleration (100) and the number of the currently executed trajectory element.

4.3 PVT Trajectories

4.3.1 Trajectory Terminology

Trajectory: continuous, multidimensional motion path. PVT stands for Position, Velocity, and Time. PVT trajectories are defined in an n-dimensional space (n = 1 to 4) and are only available with MultipleAxes groups. A PVT trajectory is generated with continuous movements of the group's positioners over several time periods. For each period, each positioner must complete a defined displacement from its current position and a defined output velocity at the end of the period. By definition, there is no constant vector velocity and no definition for a vector acceleration in contrast to Line-arc trajectories or splines.

Trajectory element (segment): An element of a PVT trajectory is defined by a set of all positioner displacements and output velocities and the duration for the segment. In the PVT data file, each element is represented by a line of values:

DT, DP1, VO1, DP2, VO2, … DPn, VOn

DT:	The segment duration in seconds.
DP1, DP2,…, DPn:	Positioners' (#1, #2,…, #n) displacements during DT.
VO1, VO2,…, VOn:	Positioners' output velocities at the end of DT.

4.3.2 Trajectory Conventions

When defining or executing a PVT trajectory, a number of rules must be followed:

- The motion group must be a MultipleAxes group.
- All trajectories must be stored in the controller's memory. Use the XPS webpage **Files → Trajectory files** to edit, upload or download a PVT trajectory file. Once a trajectory is started, it executes in the background allowing other groups to work independently and simultaneously.
- Each trajectory must have a beginning and an end. Endless (infinite) trajectories are not allowed. Although, execution of a trajectory file N-times (N defined by user) is allowed. Since the trajectory is stored in a file, the trajectory's maximum size (maximum elements number) is practically not limited.
- PVT trajectory elements (segments) are 3rd order polynomial pieces for each positioner that hit the positions P_{i-1} (at time t_{i-1} with a velocity v_{i-1}) and positions P_i

(at time t_i with a velocity v_i). There is no direct link between the trajectories of the different positioners in the group.

- PVT trajectories form a continuous path (each segment output position is equal to the next segment input position), and the segment tangential angles at the connection point of any two consecutive segments are continuous including its derivative. It means that the PVT trajectory continuity property is R^1.
- The input velocity of any element is equal to the output velocity of the previous element. The input velocity for the first element is always zero. The output velocity of the last element must be zero as well.

4.3.3 Geometric Conventions

- The coordinate system can be any convention, it does not need to be an orthogonal system.
- A PVT trajectory can be defined for a MultipleAxes group. The number of positioners in the PVT trajectory must match that of the linked with MultipleAxes group.

4.3.4 PVT Interpolation

For each positioner belonging to the MultipleAxes group, the PVT trajectory calculates a 3rd order polynomial curve P(u) that can be presented by the following equations:

Profile coefficient

- Acceleration jerk:

$$Jerk = \frac{6 \cdot \left[DT \cdot (V_{in} + V_{out}) - 2 \cdot DX \right]}{DT^3}$$

- Initial acceleration:

$$G_{in} = \frac{2 \cdot \left[3 \cdot DX - DT \cdot (2 \cdot V_{in} + V_{out}) \right]}{DT^2}$$

- Final acceleration:

$$G_{out} = \frac{2 \cdot \left[DT \cdot (V_{in} + 2 \cdot V_{out}) - 3 \cdot DX \right]}{DT^2}$$

Profile equation

- Acceleration:

$$Acc(t) = G_{in} + Jerk \cdot t$$

- Velocity:

$$Vel(t) = V_{in} + G_{in} \cdot t + \frac{Jerk \cdot t^2}{2}$$

- Position:

$$Pos(t) = V_{in} \cdot t + \frac{G_{in} \cdot t^2}{2} + \frac{Jerk \cdot t^3}{6}$$

Here:

DT is the segment duration in seconds

DX is the displacement during DT

V_{in} is the output velocity of the previous segment (which is equal to the input velocity of the current segment)

V_{out} is the output velocity of the current segment.

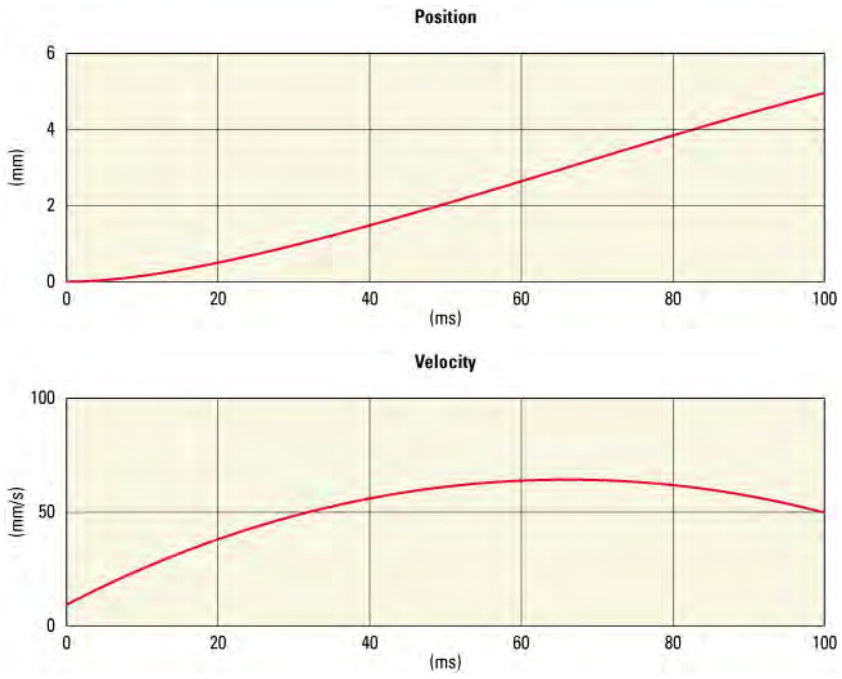
t is the time in seconds starting at 0 (entry of the current element) and ending at DT (end of the segment)

4.3.5 Influence of the Element Output Velocity to the Trajectory

The contour of each PVT trajectory element is influenced not only by the displacement, but also by the input and output velocities. As the user decides on these velocities, attention must be placed on these values to get the desired results.

The effect of the velocity is illustrated in the following example which shows the position and velocity profiles for one segment of a PVT trajectory that has a displacement of 5 mm, a duration of 100 ms, an input velocity of 10 mm/s and an output velocity of either 50 mm/s or 500 mm/s:

- If the output velocity is equal to 50 mm/s.



- If the output velocity is equal to 500 mm/s.

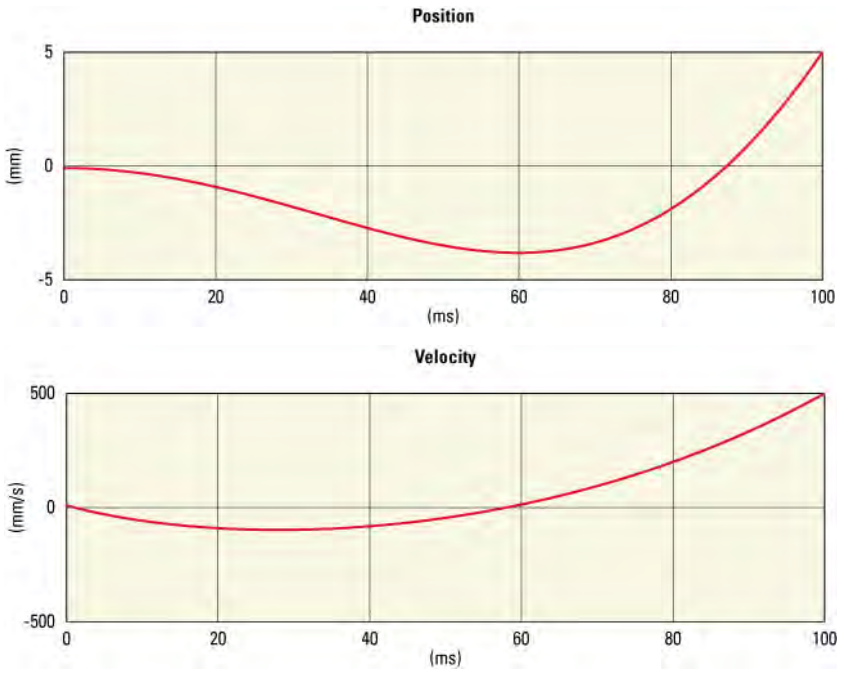


Figure 17: PVT trajectory element in execution: the comparison.

A PVT trajectory must have three parameters: position, velocity and time. With a given target displacement, output velocity and time duration, the PVT trajectory calculates intermediate positions and velocities as a function of time.

With an output velocity of 50 mm/s, the positioner has “enough” time to achieve the displacement within the assigned time (100 ms) in the forward direction. The velocity increases at the beginning and then slows down towards the end. The position always increases up to the target position (5 mm).

On the other hand, when the output velocity is set to 500 mm/s, the positioner does not have enough time to achieve the displacement and speed output required in the forward direction. So the positioner will first reverse the direction of motion to be able to approach the end position with a speed of 500 mm/s.

4.3.6 Trajectory File Description

The PVT trajectory described must be stored in the controller’s memory. Use the XPS webpage **Files → Trajectory files** to edit, upload or download a PVT trajectory file. Each line of this file represents one element of the trajectory.

A line contains several data separated by a comma. The number of data in each line depends on the number of positioners in the group. The first data in each line is the duration of the element. The following data is grouped in pairs of two representing the displacement and the output velocity for each positioner of the group. Comment lines are possible, they must be preceded with a semi-colon (“;”) character.

So the line format is as follows:

Data #1: Element duration (seconds).

Data #2: 1st positioner’s displacement (units).

Data #3: 1st positioner’s output velocity (units/s).

Data #4: 2nd positioner’s displacement (units).

Data #5: 2nd positioner’s output velocity (units/s).

(And so on…)

NOTE

The first positioner is always the first defined in the system.ini of the MultipleAxes group (see PositionerInUse), the second positioner is always defined as second, and so on…

Option to add GPIO outputs

To add either analog or digital outputs during the trajectory the first line of the file should specify the selected outputs and each element should define the GPIO output per plug with the following data:

- Digital Outputs

  For digital outputs, two values must be specified per element and per selected GPIO plug to define the state of the output at the end of the element, namely Mask and DigitalOutputValue.

- Analog Outputs

  Analog outputs are handled like position to output a controlled voltage profile. For analog outputs, two values per element and per selected output plug must be specified, the voltage variation (DU) during the element and the rate of change (VU) at the end of the element.

Example format:

; Comment: A MultipleAxes group with two positioners, one digital output and two analog outputs.

GPIO3.DO, GPIO4.DAC1, GPIO4.DAC2

DT1, DX1, DX2, Mask1, Value1, DU1, VU1, DU2, VU2

DT2, DX1, DX2, Mask1, Value1, DU1, VU1, DU2, VU2

DT3, DX1, DX2, Mask1, Value1, DU1, VU1, DU2, VU2

…

DTm, DX1, DX2, Mask1, Value1, DU1, VU1, DU2, VU2

4.3.7 Trajectory File Example

Following is an example of a PVT trajectory defined in a MultipleAxes group that contains two positioners. The tabs are added for better readability and are ignored in a line:

```
1.0,    0.4167,    1.25,    0,         0
1.0,    2.9167,    5,       0,         0
1.0,    7.0833,    8.75,    0,         0
1.0,    9.5833,    10,      0,         0
1.0,    10,        10,      0.4167,    1.25
1.0,    10,        10,      2.9167,    5
1.0,    10,        10,      7.0833,    8.75
1.0,    10,        10,      9.5833,    10
1.0,    9.5833,    8.75,    10,        10
1.0,    7.0833,    5,       10,        10
1.0,    2.91667,   1.25,    10,        10
1.0,    0.41667,   0,       10,        10
1.0,    0,         0,       9.5833,    8.75
1.0,    0,         0,       7.0833,    5
1.0,    0,         0,       2.91667,   1.25
1.0,    0,         0,       0.41667,   0
```

This file represents the following data:

Time Period (s)	Axis #1 Displacement	Axis #1 Velocity Out	Axis #2 Displacement	Axis #2 Velocity Out
1.0	0.4167	1.25	0	0
1.0	2.9167	5.0	0	0
1.0	7.0833	8.75	0	0
1.0	9.5833	10	0	0
1.0	10	10	0.4167	1.25
1.0	10	10	2.9167	5
1.0	10	10	7.0833	8.75
1.0	10	10	9.5833	10
1.0	9.5833	8.75	10	10
1.0	7.0833	5	10	10
1.0	2.9167	1.25	10	10
1.0	0.4167	0	10	10
1.0	0	0	9.5833	8.75
1.0	0	0	7.0833	5
1.0	0	0	2.9167	1.25
1.0	0	0	0.4167	0

Table 1: The trajectory data file.

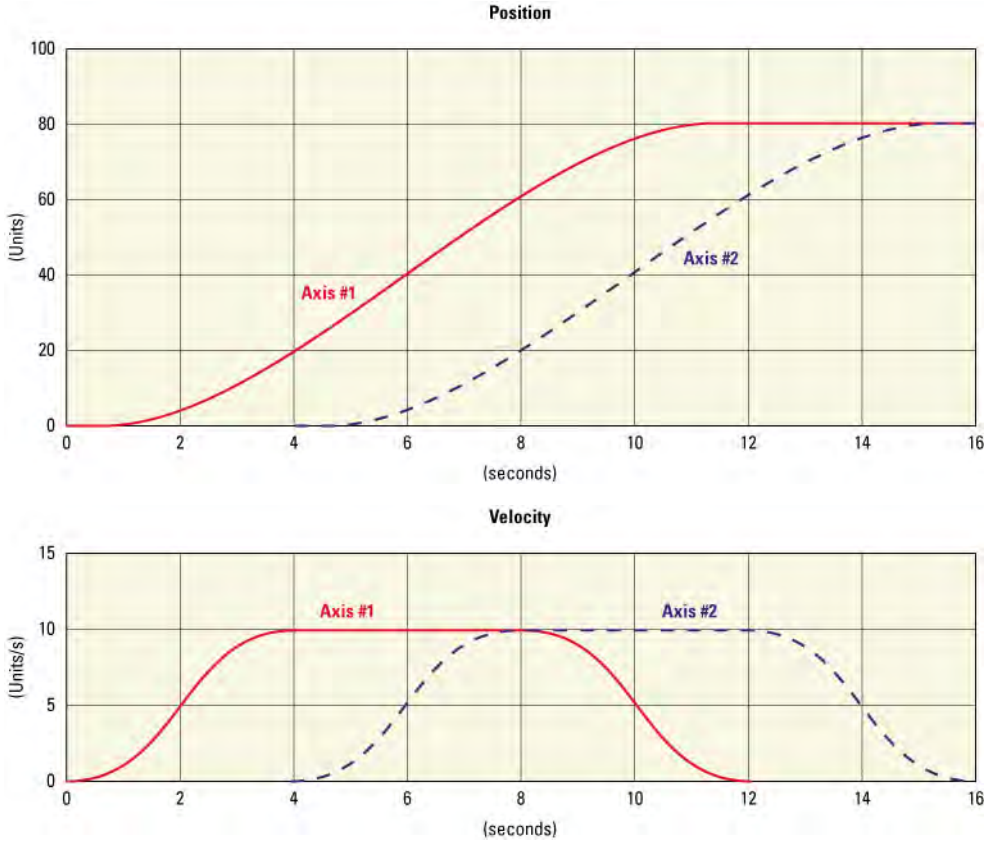


Figure 18: Executing the trajectory data file with the PVT algorithm.

4.3.8 PVT Trajectory Verification and Execution

Here are four functions to verify or execute a PVT trajectory :

- **MultipleAxesPVTVerification(): ** Verifies a PVT trajectory data file.
- **MultipleAxesPVTVerificationResultGet(): ** Returns the results of the last trajectory verification call, actuator by actuator. This function works only after a MultipleAxesPVTVerification().
- **MultipleAxesPVTExecution(): ** Executes a PVT trajectory.
- **MultipleAxesPVTParametersGet(): ** Returns the trajectory's current execution parameters. This function works only while executing a trajectory.

The function **MultipleAxesPVTVerification**() can be executed at any moment and is independent of the trajectory execution. This function does the following:

- Checks the trajectory file for data and syntax coherence.
- Simulates the trajectory to determine the positioner's travel requirements in negative and positive directions and the maximum allowed speed and acceleration for each positioner. This function determines whether the trajectory is executable.
- If all is OK, it returns an "OK" (value 0). Otherwise it returns a corresponding error. An error for instance is reported if one of the positioner's speed or acceleration reached during the trajectory exceeds the maximum allowed speed or acceleration.

The function **MultipleAxesPVTVerificationResultGet**() can be executed only after a MultipleAxesPVTVerification(). It returns the trajectory limits for each positioner, which are the travel requirements in positive and negative directions, the achieved maximum speed and acceleration.

To execute a PVT trajectory, send the function **MultipleAxesPVTExecution**() while specifying the file name and the number of cycles. This function does not verify the trajectory's coherence or geometric conditions (exceeding any positioner's min. or max. travel, speed or acceleration) before execution, so users must be careful when executing

a trajectory without verifying the trajectory first. In case of an error during execution, because of bad data or because of a following error, the motion group will make an emergency stop and will go to the disabled state.

Finally, the function MultipleAxesPVTParametersGet() returns the trajectory name and the number of the trajectory element that is currently being executed. This function returns an error if the trajectory is not executing.

4.3.9 Example with a MultpleAxes Group

MultipleAxesPVTVerification (MyGroup, PVT1.trj)

This function returns a 0 if the trajectory is executable.

MultipleAxesPVTVerificationResultGet (MyGroup.MyPositioner1, *Name, *NegTravel, *PosTravel, *MaxSpeed, *MaxAcceleration)

This function returns the name of the trajectory verified with the last functions call of MultipleAxesPVTVerification to the motion group MyGroup(PVT1.trj) and the trajectory limits for the positioner MyGroup.MyPositioner1. These trajectory limits are: the negative or left travel requirement, the positive or right travel requirement, the achieved maximum speed and acceleration. Make sure that these trajectory limits (required negative and positive travel, speed and acceleration) are within the soft limits of the stages defined in the stages.ini file (section Travel: MinimumTargetPosition, MaximumTargetPosition and section Profiler: MaximumVelocity, MaximumAcceleration).

MultipleAxesPVTExecution (MyGroup, PVT1.trj, 5)

Executes the trajectory PVT1.trj five (5) times.

MultipleAxesPVTParametersGet (MyGroup, *FileName, *ElementNumber)

Returns the currently executed trajectory file name (PVT1.trj) and the number of the currently executed trajectory element.

4.4 PT Trajectrories

4.4.1 Trajectory Terminology

Trajectory: continuous, multidimensional motion path. PT stands for Position and Time. PT trajectories are defined in an n-dimensional space (n = 1 to 4) and are available only with MultipleAxes groups. A PT trajectory is generated with continuous movements of the group's positioners over several time periods. For each period, each positioner must complete a defined displacement from its current position. By definition, there is no constant vector velocity and no definition for vector acceleration compared to Line-arc trajectories or splines.

Trajectory element (segment): An element of a PT trajectory is defined by a set of all positioner displacements and the duration for the segment. In the PT data file, each element is represented by a line of values:

DT, DP1, DP2, … DPn

DT: The segment duration in seconds.

DP1, DP2,…, DPn: Positioners' (#1, #2,…, #n) displacements during DT.

4.4.2 Trajectory Conventions

When defining or executing a PT trajectory, a number of rules must be followed:

- The motion group must be a MultipleAxes group.
- All trajectories must be stored in the controller's memory. Use the XPS webpage **Files → Trajectory files** to edit, upload or download a PT trajectory file. Once a trajectory is started, it executes in the background allowing other groups to work independently and simultaneously.

- Each trajectory must have a beginning and an end. Endless (infinite) trajectories are not allowed. Although, execution of a trajectory file N-times (N defined by user) is allowed. Since the trajectory is stored in a file, the trajectory's maximum size (maximum elements number) is practically not limited.
- PT trajectory elements (segments) are 3rd order polynomial pieces for each positioner that hit the positions P_{i-1}, at time t_{i-1}, and positions P_i, at time t_i. There is no direct link between the trajectories of the different positioners in the group.
- PT trajectories form a continuous path: each segment output position is equal to the next segment input position and the input velocity of any element is equal to the output velocity of the previous element. Hence the segment tangential angles at the connection point of any two consecutive segments are continuous including its derivative. It means that the PT trajectory continuity property is R^1.
- The input velocity for the first element is always zero. The output velocity of the last element must be zero as well.

4.4.3 Geometric Conventions

- The coordinate system can be any convention; it does not need to be an orthogonal system.
- A PT trajectory can be defined for a MultipleAxes group. The number of positioners in the PT trajectory must match that of the linked with MultipleAxes group.

4.4.4 PT Interpolation

For each positioner belonging to the MultipleAxes group, the PT trajectory calculates a 3rd order polynomial curve X(t) that can be represented by the following equations:

$$Current\ element = (DT1, DX1)$$

$$Next\ element = (DT2, DX2)$$

$$X_{in} = X_{out}\ (previous)$$

$$V_{in} = V_{out}\ (previous)$$

$$V_{out} = \frac{DX2 \cdot DT1^2 + DX1 \cdot DT2^2}{DT1 \cdot DT2 \cdot (DT1 + DT2)}$$

$$G_{in} = \frac{2 \cdot [3 \cdot DX1 - DT1 \cdot (2 \cdot V_{in} + V_{out})]}{DT1^2}$$

$$Jerk = \frac{6 \cdot (DT1 \cdot V_{in} - 2 \cdot DX1 + DT1 \cdot V_{out})}{DT1^3}$$

$$X(t) = X_{in} + V_{in} \cdot t + \frac{1}{2} \cdot G_{in} \cdot t^2 + \frac{1}{6} \cdot Jerk \cdot t^3$$

The algorithm is the same as for PVT mode except that the output velocity is not set in the trajectory file, but calculated by the firmware using the following rule:

The crossing velocity of a point is defined by taking into account the previous and the following point, as if the three points where to be crossed with a constant acceleration. This crossing velocity becomes the set output velocity of the element defined by the first two points. The result is a lower speed ripple than a path at constant acceleration.

4.4.5 Trajectory File Description

The PT trajectory described must be stored in the controller's memory. Use the XPS webpage **Files → Trajectory files** to edit, upload or download a PT trajectory file. Each line of this file represents one element of the trajectory.

A line contains several data entries separated by a comma. Comment lines are possible, and must be preceded with a semi-colon (";") character. The number of data entries in each line depends on the number of positioners in the MultipleAxes group. The first

data entry in each line is the duration of the element. The following data entries represent the displacement for each positioner of the group.

So a generic format is as follows:

DT1, DX1, DX2, …, DXn
DT2, DX1, DX2, …, DXn
…
DTm-2, DX1, DX2, …, DXn
DTm-1, 0, 0, …, 0
DTm, 0, 0, …, 0

NOTES

To guarantee that the output velocity is null at the end of PT trajectory execution, two data lines with zero displacements must be present at the end of the PT trajectory.

The first positioner is always the first defined in the system.ini of the MultipleAxes group (see PositionerInUse), the second positioner is always defined as second, and so on…

Option to add GPIO outputs

To add either analog or digital outputs during the trajectory the first line of the file should specify the selected outputs and each element should define the GPIO output per plug with the following data:

- Digital Outputs

  For digital outputs, two values must be specified per element and per selected GPIO plug to define the state of the output at the end of the element, namely Mask and DigitalOutputValue.

- Analog Outputs

  Analog outputs are handled like position to output a controlled voltage profile. For analog outputs, one value per element and per output plug must be specified, the voltage variation (DU) during the element.

Example format:

; Comment: A MultipleAxes group with two positioners, one digital output and two analog outputs

GPIO3.DO, GPIO4.DAC1, GPIO4.DAC2
DT1, DX1, DX2, Mask1, Value1, DU1, DU2
DT2, DX1, DX2, Mask1, Value1, DU1, DU2
DT3, DX1, DX2, Mask1, Value1, DU1, DU2
…
DTM, DX1, DX2, Mask1, Value1, DU1, DU2

4.4.6 Trajectory File Example

Following is an example of a PT trajectory defined in a MultipleAxes group that contains two positioners and configured to output one digital and one analog output. The data entries in a line are separated by a comma (","). Comment lines are possible and must be preceded with a semi-colon (";") character. Because of the PT trajectory internal calculation of elements end velocity, two lines with zero displacements must be present at the end of the PT trajectory file to guarantee that the elements end velocity be zero at the end of trajectory execution. In the example below tabs were added for better readability:

; PT trajectory data					
GPIO3.DO,	GPIO4.DAC1,				
1.0,	0.4167,	0,	65535,	1,	0.04167
1.0,	2.9167,	0,	65535,	2,	0.29167
1.0,	7.0833,	0,	65535,	4,	0.70833
1.0,	9.5833,	0,	65535,	8,	0.95833
1.0,	10,	0.4167,	65535,	16,	1.0
1.0,	10,	2.9167,	65535,	32,	1.0
1.0,	10,	7.0833,	65535,	64,	1.0
1.0,	10,	9.5833,	65535,	128,	1.0
1.0,	9.5833,	10,	65535,	64,	0.95833
1.0,	7.0833,	10,	65535,	32,	0.70833
1.0,	2.9167,	10,	65535,	16,	0.29167
1.0,	0.4167,	10,	65535,	8,	0.04167
1.0,	0,	9.5833,	65535,	4,	0
1.0,	0,	7.0833,	65535,	2,	0
1.0,	0,	2.9167,	65535,	1,	0
1.0,	0,	0.4167,	65535,	0,	0
1.0,	-0.4167,	0,	65535,	1,	-0.04167
1.0,	-2.9167,	0,	65535,	2,	-0.29167
1.0,	-7.0833,	0,	65535,	4,	-0.70833
1.0,	-9.5833,	0,	65535,	8,	-0.95833
1.0,	-10,	-0.4167,	65535,	16,	-1.0
1.0,	-10,	-2.9167,	65535,	32,	-1.0
1.0,	-10,	-7.0833,	65535,	64,	-1.0
1.0,	-10,	-9.5833,	65535,	128,	-1.0
1.0,	-9.5833,	-10,	65535,	64,	-0.95833
1.0,	-7.0833,	-10,	65535,	32,	-0.70833
1.0,	-2.9167,	-10,	65535,	16,	-0.29167
1.0,	-0.4167,	-10,	65535,	8,	-0.04167
1.0,	0,	-9.5833,	65535,	4,	0
1.0,	0,	-7.0833,	65535,	2,	0
1.0,	0,	-2.9167,	65535,	1,	0
1.0,	0,	-0.4167,	65535,	0,	0

This file represents the following data:

GPIO #3 Digital Output	GPIO #4 Analog Output
GPIO3.DO	GPIO4.DAC1

Time Period (S)	Axis #1 Displacement	Axis #2 Displacement	GPIO #3 Mask1	GPIO #3 Value 1	GPIO #4 Analog voltage variation DU2
1.0	0.4167	0	65535	1	0.04167
1.0	2.9167	0	65535	2	0.29167
1.0	7.0833	0	65535	4	0.70833
1.0	9.5833	0	65535	8	0.95833
1.0	10	0.4167	65535	16	1.0
1.0	10	2.9167	65535	32	1.0
1.0	10	7.0833	65535	64	1.0
1.0	10	9.5833	65535	128	1.0
1.0	9.5833	10	65535	64	0.95833
1.0	7.0833	10	65535	32	0.70833
1.0	2.9167	10	65535	16	0.29167
1.0	0.4167	10	65535	8	0.04167
1.0	0	9.5833	65535	4	0
1.0	0	7.0833	65535	2	0
1.0	0	2.9167	65535	1	0
1.0	0	0.4167	65535	0	0
1.0	-0.4167	0	65535	1	-0.04167
1.0	-2.9167	0	65535	2	-0.29167
1.0	-7.0833	0	65535	4	-0.70833
1.0	-9.5833	0	65535	8	-0.95833
1.0	-10	-0.4167	65535	16	-1.0
1.0	-10	-2.9167	65535	32	-1.0
1.0	-10	-7.0833	65535	64	-1.0
1.0	-10	-9.5833	65535	128	-1.0
1.0	-9.5833	-10	65535	64	-0.95833
1.0	-7.0833	-10	65535	32	-0.70833
1.0	-2.9167	-10	65535	16	-0.29167
1.0	-0.4167	-10	65535	8	-0.04167
1.0	0	-9.5833	65535	4	0
1.0	0	-7.0833	65535	2	0
1.0	0	-2.9167	65535	1	0
1.0	0	-0.4167	65535	0	0

4.4.7 PT Trajectory Verification and Execution

Here are four functions to verify or execute a PT trajectory :

- **MultipleAxesPTVerification**(**):** Verifies a PT trajectory data file.
- **MultipleAxesPTVerificationResultGet**(**):** Returns the results of the last trajectory verification call, positioner by positioner. This function works only after a MultipleAxesPTVerification().
- **MultipleAxesPTExecution**(**):** Executes a PT trajectory.
- **MultipleAxesPTParametersGet**(**):** Returns the trajectory's current execution parameters. This function works only while executing a trajectory.

The function **MultipleAxesPTVerification**() can be executed at any moment and is independent of the trajectory execution. This function does the following:

- Checks the trajectory file for data and syntax coherence.
- Simulates the trajectory to determine the positioner's travel requirements in negative and positive directions and the maximum allowed speed and acceleration for each positioner. This function determines whether the trajectory is executable.
- If all is OK, it returns an "OK" (value 0). Otherwise it returns a corresponding error. An error for instance is reported if one of the positioner's speed or acceleration reached during the trajectory exceeds the maximum allowed speed or acceleration.

The function **MultipleAxesPTVerificationResultGet**() can be executed only after a MultipleAxesPTVerification(). It returns the trajectory limits for each positioner, which are the travel requirements in positive and negative directions, the achieved maximum speed and acceleration.

To execute a PT trajectory, send the function **MultipleAxesPTExecution**() while specifying the file name and the number of cycles. This function does not verify the trajectory's coherence or geometric conditions (exceeding any positioner's min. or max. travel, speed or acceleration) before execution, so users must be careful when executing a trajectory without verifying the trajectory first. In case of an error during execution, because of bad data or because of a following error, the motion group will make an emergency stop and will go to the disabled state.

Finally, the function **MultipleAxesPTParametersGet**() returns the trajectory name and the number of the trajectory element that is currently being executed. This function returns an error if the trajectory is not executing.

4.4.8 Example of how to use PVT functions

MultipleAxesPTVerification (MultipleGroup, PTExample.trj)

This function returns a 0 if the trajectory is executable.

MultipleAxesPTVerificationResultGet (MultipleGroup.Pos1, *Name, *NegTravel, *PosTravel, *MaxSpeed, *MaxAcceleration)

This function returns the name of the trajectory verified with the last functions call of MultipleAxesPTVerification to the motion group MultipleGroup (PTExample.trj) and the trajectory limits for the positioner MultipleGroup.Pos1. These trajectory limits are: the negative or left travel requirement, the positive or right travel requirement, the achieved maximum speed and acceleration. Make sure that these trajectory limits (required negative and positive travel, speed and acceleration) are within the soft limits of the stages defined in the stages.ini file (section Travel: MinimumTargetPosition, MaximumTargetPosition and section Profiler: MaximumVelocity, MaximumAcceleration).

MultipleAxesPTExecution (MultipleGroup, PTExample.trj, 5)

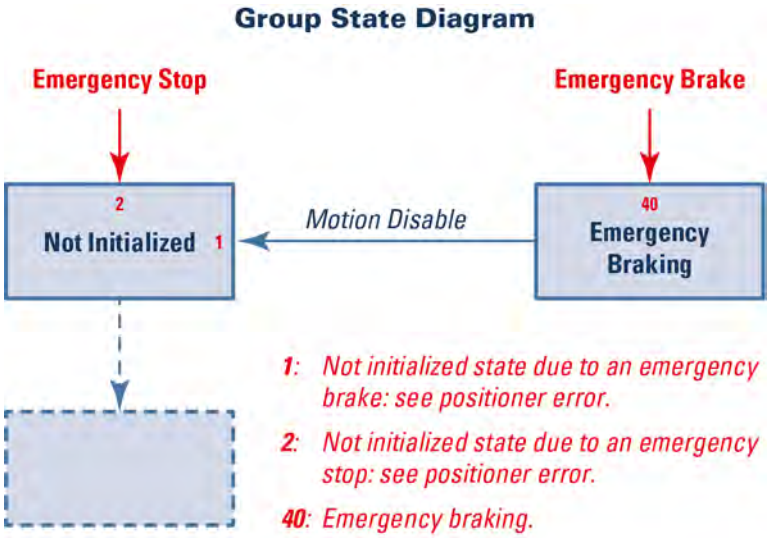
This function executes the trajectory, PTExample.trj, five times.

MultipleAxesPTParametersGet (MultipleGroup, *FileName, *ElementNumber)

Returns the currently executed trajectory file name (PTExample.trj) and the number of the currently executed trajectory element.

5.0 Emergency Brake and Emergency Stop Cases

5.1 Principle



NOTE

Emergency brake brings a stage to a stop, then sets the motor power to Off.
Emergency stop: sets motor power to Off only.

5.2 Emergency Brake Cases

Case	Error
Standard end of run driver safety supervisor Standard limit and home encoder safety supervisor Standard limit and limit encoder safety supervisor	• Plus end of run is detected • Minus end of run is detected
Line arc trajectory execution	• Error occurs when reading or getting trajectory parameters • The user target position is outside the MinimumTargetPosition and MaximumTargetPosition value • Actual positioner velocity is greater than the *MaximumVelocity* value
Spline trajectory execution	• Error occurs when reading or getting trajectory parameters • The user target position is outside the *MinimumTargetPosition* and *MaximalTargetPosition* value • Actual positioner velocity is greater than the *MaximumVelocity* value
PVT trajectory execution	• Error occurs when reading or getting trajectory parameters • Error occurs during trajectory execution • The user target position is outside the *MinimumTargetPosition* and *MaximalTargetPosition* values • Actual positioner velocity is greater than the *MaximumVelocity* value
S-gamma motion of a slave	• Group positioner is not in the home process, • **And** end of run detection is enabled • **And** the group is not a spindle group • **And** the user target position is outside the *MinimumTargetPosition* and *MaximalTargetPosition* value

5.3 Emergency Stop Cases

Case	Error
AquadBEncoder fault	• Quadrature error • FOC fault (over run error)
Analog interpolator encoder fault	• Quadrature error • FOC fault • Sin Cos radius error
AnalogAccelerationMotorInterface AnalogDualSinAccelerationMotorInterface AnalogPositionMotorInterface AnalogSinAccelerationMotorInterface AnalogStepperPositionMotorInterface AnalogVelocityMotorInterface AnalogVoltageMotorInterface DigitalStepperPositionMotorInterface AnalogSinAccelerationLMIMotorInterface AnalogAccelerationTZMotorInterface AnalogPositionPiezoMotorInterface	• Driver fault
Single Axis with clamping control Single Axis theta	• Unclamped state

6.0 Compensation

6.1 Definitions

The XPS controller features different compensation methods that improve the performance of a motion system namely, Backlash, Linear error, Positioner mapping, XY mapping and XYZ mapping. To understand the different compensation methods it is important to define terms used to calculated the compensation.

TargetPosition: The TargetPosition is the position where the positioner must be after the completion of a move.

SetpointPosition : The SetpointPosition is the theoretical position commanded to the servo loop. It is the position where the positioner should be, during and after the end of the move.

CurrentPosition: The CurrentPosition is the current physical position of the positioner. It is equal to the encoder position after all compensations (backlash, linear error and mapping) have been taken into account.

SetpointVelocity: The SetpointVelocity is calculated by the motion profiler and represents the “theoretical” velocity to reach during the motion.

SetpointAcceleration: The SetpointAcceleration is calculated by the motion profiler and represents the “theoretical” acceleration to reach during the motion.

FollowingError: The FollowingError is the difference between the CurrentPosition and the SetpointPosition.

A short description of the different compensation methods that improves the performance of a motion system follows:

Backlash compensation: The use of backlash compensation improves the bi-directional repeatability and accuracy of a motion device that has mechanical play. Backlash compensation is applicable to all positioners, but it is not available in all motion modes. When backlash compensation is activated, the XPS controller adds a user-defined BacklashValue to the TargetPosition to calculate a new target position whenever the direction of motion reverses. This internally used new target position is then the basis for the calculations of the motion profiler. No modification of the actual target is performed.

Linear error compensation: The linear error compensation helps improve the accuracy of a motion device by eliminating linear error sources. Linear errors can be caused by screw pitch errors, linear increasing angular deviations (abbe errors), thermal effects or cosine errors (misalignment between the feedback device and the direction of motion). Linear error compensation is applicable to all positioners. Its value is defined in the stages.ini. When set to other than zero, the encoder positions are compensated by this value. Linear error compensation can be used in conjunction with other compensation. For this reason, keep in mind the effects of using linear error compensation in addition to other compensation methods.

Positioner mapping: In contrast to the linear error compensation, positioner mapping also allows compensation for nonlinear error sources. Positioner mapping is done by sending a compensation table to the XPS controller and configuring the needed settings in the stages.ini. Positioner mapping is available with all positioners and works in parallel with other compensations except for the backlash compensation method. Better accuracy performance is achievable with linear compensation and positioner mapping combined.

XY mapping: XY mapping is only available with XY groups. It allows compensation for all errors of an XY group at any position of the XY group by sending two compensation tables to the XPS controller (x and y compensations mapped to x and y positions). The XY mapping is dynamically taken into account on the corrector loop of the XPS controller. XY mapping works in parallel to other compensation methods.

Keep in mind that the results of XY mapping may not be the same as those of Positioner mapping or linear compensation alone.

XYZ mapping: XYZ mapping is only available with XYZ groups. It compensates for all errors of an XYZ group at any position of the XYZ group by sending three compensation files to the XPS controller (x compensations mapped to x, y, and z positions, and so on). The XYZ mapping is dynamically taken into account on the corrector loop of the XPS controller. XYZ mapping works in parallel to other compensation methods. Keep in mind that the results of XYZ mapping may not be the same as those of Positioner mapping or linear compensation alone.

TargetPosition, SetpointPosition & CurrentPosition are accessible via function and Gathering (Data Collection).

SetpointVelocity, SetpointAcceleration & FollowingError are accessible via Gathering (Data Collection).

6.2 Backlash Compensation

Backlash compensation is applicable on all positioners, but works only under certain conditions:

- The “HomeSearchSequenceType” in the stages.ini must be different from “CurrentPositionAsHome”.
- Backlash compensation is not compatible with positioner mapping. So for positioners with backlash compensation, it is not allowed to have an entry for “PositionerMappingFileName” in the stages.ini.
- Backlash compensation is not compatible with trajectories (Line-Arc, Spline, PVT), jog or analog tracking. So it is not possible to execute any trajectory, to use the jog mode or to enable the analog tracking with any motion group that contains positioners with backlash compensation enabled.

After the above has been taken into consideration, a number of steps need to be taken to enable backlash compensation. First of all, there must be a value larger than 0 for “backlash” in the stages.ini. But this setting does not automatically enable backlash compensation. To do so, send the function **PositionerBacklashEnable**() while the motion group, which includes the positioner is disabled. To disable backlash compensation (for instance to execute a jog motion or to use analog tracking), use the function **PositionerBacklashDisable**()**.** The value for backlash compensation can be changed at any time with the function **PositionerBacklashSet**()**.** The new value for the backlash will be taken into account with the next following move. Finally, the function **PositionerBacklashGet**() returns the current value of the backlash and the backlash status (“enabled” or “disabled”).

For backlash setting to remain set after power down, the stages.ini file must be modified with the value desired.

Example

In the Backlash section of the stages.ini file, set a value greater than or equal to 0:

```
;--- Backlash
Backlash = 5                    ; units
```

This example shows the sequence of functions that enable backlash compensation:

PositionerBacklashEnable (MyGroup.MyPositioner)

GroupInitialize (MyGroup)

GroupHomeSearch (MyGroup)

…

PositionerBacklashSet (MyGroup.MyPositioner, 10)

PositionerBacklashGet (MyGroup.MyPositioner, *Backlash, *Status)

Returns the backlash value (10) and the backlash status (Enable).

…

PositionerBacklashDisable (MyGroup.MyPositioner)

6.3 Linear Error Correction

Linear error correction is applicable on all positioners and works in parallel with any other compensation. To use linear error correction, you need to set a value for "LinearErrorCorrection" in the stages.ini. When set, the corrected positions are calculated in the following way:

Corrected position = EncoderPosition x (1 + LinearEncoderCorrection/10^6)

The value of LinearEncoderCorrection is specified in ppm (parts per million). The correction is applied relative to the physical home position of the positioner (the Encoder position by definition is set to the HomePreset value at the home position). This hardware reference for linear error correction has the advantage of being independent of the value of the HomePreset.

Example

In the Encoder section of the stages.ini file, set a value other than 0, but $-0.5 \times 10^6 < \text{value} < 0.5 \times 10^6$, in parameter LinearEncoderCorrection:

```
;--- Encoder
EncoderType =AquadB
EncoderResolution = 0.001       ; unit
LinearEncoderCorrection =5      ; ppm
```

6.4 Positioner Mapping

Positioner mapping corrects for any nonlinear errors of a positioner. Positioner mapping is applicable on all positioners and can be used with other compensations except backlash compensation. The positioner mapping is applied after linear correction is done.

The positioner mapping data is defined in a text file. Each line of that file represents one set of data. Each set of data is composed of the position and the error at this position. The separator between the two data entries in each line is a tab. All positions are relative to the physical home position of the positioner. The data file must contain the line "0 0", which means that the error at the home position is 0. This hardware reference for positioner mapping has the advantage of being independent of the value of the HomePreset.

The following shows the general structure of such a data file:

```
PosMin      Error 0
Pos 1       Error 1
Pos 2       Error 2
…           …
0           0
…           …
PosMax              Error LineNumber-1
```

To activate positioner mapping, the mapping file must be in the ..\admin\config directory of the XPS controller and the following settings must be configured in the stages.ini:

- **PositionerMappingFileName:** Name of the mapping file.
- **PositionerMappingLineNumber:** Number of lines of the file.
- **PositionerMappingMaxPositionError:** Maximum absolute error in the file must be larger than any entry in the mapping file. To be read properly, the error entries must be in index format, see example.

PositionerMappingLineNumber and PositionerMappingMaxPositionError are only used to check for the correctness of the mapping file.

Example

The following shows an example of a positioner mapping data file:
PosMapping.txt

-3.00	-0.00125
-2.00	-0.00112
-1.00	-0.00137
0.00	0.00000
1.00	0.00140
2.00	0.00145
3.00	0.00154

Define the positioner mapping in the stages.ini file:

```
;--- Backlash
Backlash =0                    ; unit

;--- Positioner mapping
PositionerMappingFileName = PosMapping.txt
PositionerMappingLineNumber = 7
PositionerMappingMaxPositionError = 0.00154

;--- Travels
MinimumTargetPosition =-3      ; unit
HomePreset =0                  ; unit
MaximumTargetPosition =3       ; unit
```

NOTE

These travel limits must be equal to or be within the positioner's limit positions of the mapping file (+3 and -3 in the above example).

Use of the functions:

- GroupInitialize(MyGroup)
- GroupHomeSearch(MyGroup)
- GroupMoveAbsolute(MyGroup.Positioner, 0.25)

The mapping file must at least cover the minimum and the maximum travel of the positioner. It must cover MinimumTargetPosition and MaximumTargetPosition parameters defined in the stages.ini, section Travels. In the example above, the travel of the positioner can not be larger than ±3 units, but it can be smaller than this. The units for the data are the same as defined by EncoderResolution in the stages.ini. The data reads as follows: the corrected position at position 3.00 units is 2.99846 units (3.00 - 0.00154). Between two data points, the XPS controller performs a linear interpolation of

the error. The corrected position at position 0.25 units is 0.24965 units (0.25 - 0.00140*0.25/1).

NOTE

Mapping is a function implemented within the controller to correct positioning errors. Once activated, mapping is transparent to the user. The function GroupPositionCurrentGet doesn't return 0.24965 (0.25 - 0.00140*0.25/1) but 0.25.

6.5 XY Mapping

XY mapping is only available to XY groups. It compensates for all errors of an XY group at any position of that XY group. XY mapping can be used in addition to other compensations, including positioner mapping. So care must be taken about the unwanted cross-effects of using XY mapping and other compensation at the same time.

XY mapping is defined by 2 compensation tables, in text file format, each for X and Y errors. In each of these files:

- The first entry in that file must be 0 (zero).
- The **first column** specifies the **X positions** (X being the first positioner of the XY group).
- The **first row** specifies the **Y positions**.
- Each cell represents the error for that X, Y position.
- The **separator** between the different data in each row is the **tab**.
- All positions are relative to the physical home position of the XY group.
- The data files must contain the X position = 0 and the Y position = 0.
- The **error at X = Y = 0 must be 0**, which means that the error at the home position is 0.

This hardware reference for XY mapping has the advantage of being independent of the value of the HomePreset.

The following shows the structure of such mapping files:

X Mapping File

XMappingColumnNumber

XMappingLineNumber

0	Y min.	Y1	…	0	…	Y max.
X min.	X err 0 0	…	…	…	…	…
X1	…	…	…	…	…	…
…	…	…	…	…	…	…
0	…	…	…	0	…	…
…	…	…	…	…	…	…
X max.	…	…	…	…	…	…

Y Mapping File

YMappingColumnNumber

YMappingLineNumber

0	Y min.	Y1	…	0	…	Y max.
X min.	Y err 0 0	…	…	…	…	…
X1	…	…	…	…	…	…
…	…	…	…	…	…	…
0	…	…	…	0	…	…
…	…	…	…	…	…	…
X max.	…	…	…	…	…	…

Figure 19: XY mapping files.

NOTE

Error in X = Y = 0 must be 0. This value in the file corresponds to the HomePreset position in the XY group reference.

To activate XY mapping, the mapping files must be in the ..\admin\config directory of the XPS controller and the following settings must be configured in the **system.ini**:

- **XMappingFileName:** Name of the mapping file.
- **XMappingLineNumber:** Total number of lines of that file.
- **XMappingColumnNumber:** Total number of columns of that file.

- **XMappingMaxPositionError:** Maximum absolute error in that file as shown in the tables below (any value larger than the actual largest value in that file will be accepted as well).
- **YMappingFileName:** Name of the mapping file.
- **YMappingLineNumber:** Total number of lines of that file.
- **YMappingColumnNumber:** Total number of columns of that file.
- **YMappingMaxPositionError:** Maximum absolute error in that file (any value larger than the actual largest value in that file will be accepted as well).

The X(Y)MappingLineNumber, X(Y)MappingColumnNumber and X(Y)MappingMaxPositionError are only used to check for the correctness of the mapping file.

Example

The following shows an example of the X and Y mapping files:

Matrix X: XYMapping_X.txt

0	-3.00	-2.00	-1.00	0.00	1.00	2.00	3.00
-3.00	-0.00192	-0.00534	-0.00254	0.00023	0.00254	0.00534	0.00192
-2.00	-0.00453	-0.00322	-0.00676	0.00049	0.00676	0.00322	0.00453
-1.00	-0.00331	-0.00845	-0.00769	0.00102	0.00769	0.00845	0.00331
0.00	-0.00787	-0.00228	-0.00787	0	0.00787	0.00228	0.00787
1.00	-0.00232	-0.00210	-0.00342	0.00089	0.00342	0.00210	0.00232
2.00	-0.00134	-0.00308	-0.00675	0.00101	0.00675	0.00308	0.00134
3.00	-0.00789	-0.00148	-0.00234	0.00121	0.00234	0.00148	0.00789

Matrix Y: XYMapping_Y.txt

0	-3.00	-2.00	-1.00	0.00	1.00	2.00	3.00
-3.00	-0.00172	-0.00434	-0.00154	0.00013	0.00204	0.00234	0.00122
-2.00	-0.00433	-0.00222	-0.00376	0.00029	0.00636	0.00222	0.00353
-1.00	-0.00311	-0.00635	-0.00569	0.00089	0.00739	0.00245	0.00231
0.00	-0.00737	-0.00128	-0.00387	0	0.00567	0.00128	0.00387
1.00	-0.00212	-0.00110	-0.00142	0.00079	0.00332	0.00310	0.00132
2.00	-0.00114	-0.00208	-0.00375	0.00089	0.00375	0.00348	0.00122
3.00	-0.00689	-0.00128	-0.00134	0.00101	0.00232	0.00138	0.00689

Verify in the stages.ini for both stages:

```
;--- Travels
MinimumTargetPosition =-3      ; unit
HomePreset =0; unit
MaximumTargetPosition =3       ; unit
```

NOTE

The limit travels must be equal or within the X and Y limit positions of the mapping files, +3 and –3, respectively in this example.

Apply the following settings in the system.ini file:

```
;--- Mapping XY
XMappingFileName = XYMapping_X.txt
XMappingLineNumber = 7
XMappingColumnNumber = 7
XMappingMaxPositionError = 0.00845

YMappingFileName = XYMapping_Y.txt
YMappingLineNumber = 7
YMappingColumnNumber = 7
YMappingMaxPositionError = 0.00739
```

Use of the functions:

- GroupInitialize(XY)
- GroupHomeSearch(XY)
- GroupMoveAbsolute(XY, 3, 2)

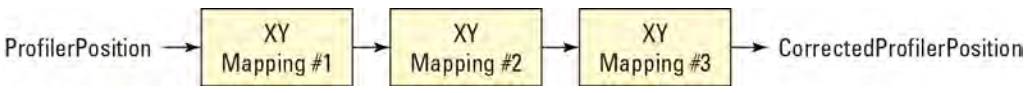
The mapping files must at least cover the minimum and the maximum travel of the XY group (they must cover the MinimumTargetPosition and the MaximumTargetPosition for the X and Y positioners, parameters defined in the stages.ini, see section Travels). So in the above example, the travel of the X and Y positioners can not be larger than ±3 units, but they can be smaller than this. The units for the data are the same as defined by the EncoderResolution in the stages.ini. The data reads as follows: at position X = 3.00 units, Y = 2.00 units the corrected X position is 2.99852 units (3.00 - 0.00148) and the corrected Y position is 1.99862 units (2.00 - 0.00138). Between two data points, the XPS controller performs a linear interpolation of the error. The two mapping files don't need to contain the same X and Y positions.

NOTE

Mapping is a function implemented within the XPS controller to correct positioning errors. When mapping is activated, it is transparent to the user. At position (X,Y) = (3.00, 2.00), the function GroupPositionCurrentGet(XY.X) doesn't return 2.99852 (3.00 - 0.00148) but 3.

6.5.1 Multiple XY Mappings in Series

Up to three XY mapping parameters are permitted for an XY group only for PP Firmware Version.



Below is the mapping section from system.ini for three XY mappings in series.

```
[GroupXY]

…

;--- Mapping XY #1
XMappingFileName =
XMappingColumnNumber=
XMappingLineNumber=
```

```
XMappingMaxPositionError=

YMappingFileName =
YMappingColumnNumber=
YMappingLineNumber=
YMappingMaxPositionError=

;--- Mapping XY #2 (PP only)
XMapping2FileName=
XMapping2ColumnNumber=
XMapping2LineNumber=
XMapping2MaxPositionError=

YMapping2FileName=
YMapping2ColumnNumber=
YMapping2LineNumber=
YMapping2MaxPositionError=

;--- Mapping XY #3 (PP only)
XMapping3FileName=
XMapping3ColumnNumber=
XMapping3LineNumber=
XMapping3MaxPositionError=

YMapping3FileName=
YMapping3ColumnNumber=
YMapping3LineNumber=
YMapping3MaxPositionError=
```

6.6 XYZ Mapping

XYZ mapping is available only with XYZ groups. It compensates for all errors of an XYZ group at any position of that XYZ group. XYZ mapping can be used in conjunction with other compensations, including positioner mapping. Care must be taken to consider the effects when using XYZ mapping and other compensations at the same time.

XYZ mapping is defined by 3 compensation files (compensation for errors in X, Y or Z), in text format. Each of these files can be seen as the juxtaposition of successive tables where the first column of the first table contains the X positions; the first row of the first table contains the Y positions; and the first cell of each table contains one of the Z positions. Each table represents a plane defined by the Z position of the first cell. The separator between the different data in each row is a tab. For legibility, inserting an empty line between successive tables is recommended, but not mandatory. The other cells contain the corresponding error.

All positions are relative to the physical home position of the XYZ group. The data files must contain the X position = 0, the Y position = 0, and the Z position = 0. The error at X = Y = Z = 0 must be 0, which means that the error at the home position is 0. This hardware reference for XYZ mapping has the advantage of being independent of the value of the HomePreset.

Figure 20 shows the structure for the three mapping files for X, Y, and Z corrections:

- **XYZMappingCorrectionX.dat:** All Err entries are X errors (corrections for X).
- **XYZMappingCorrectionY.dat:** All Err entries are Y errors (corrections for Y).
- **XYZMappingCorrectionZ.dat:** All Err entries are Z errors (corrections for Z).

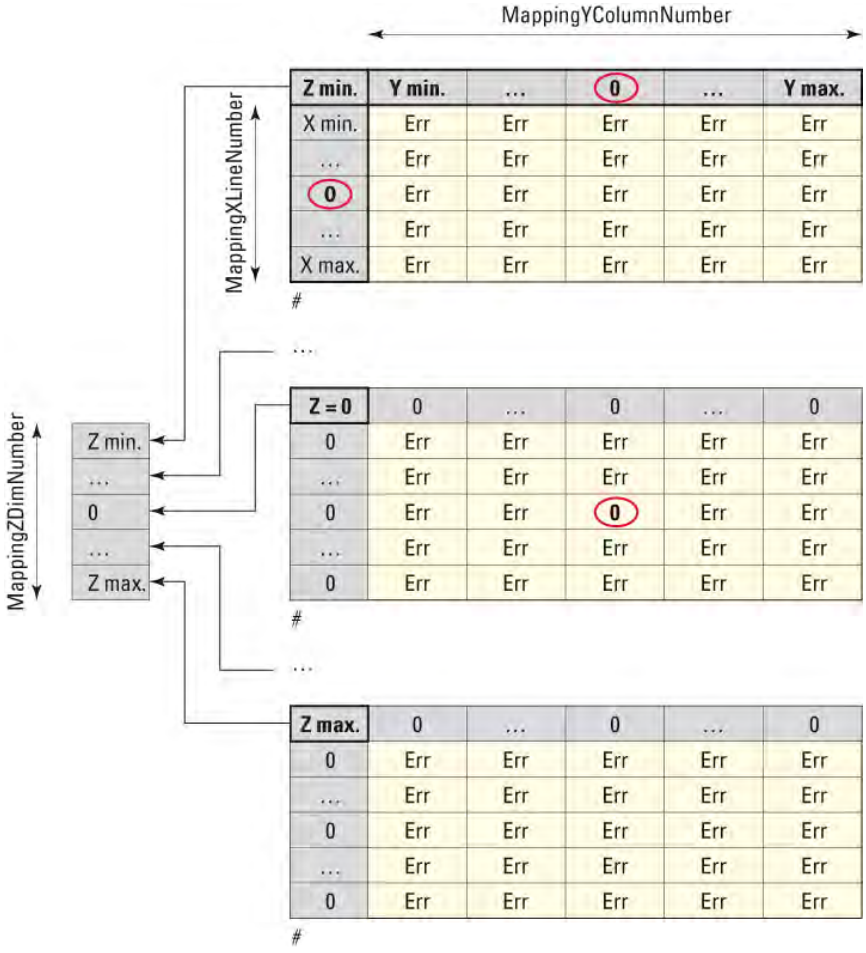


Figure 20: XYZ mapping files.
Error in each compensation file can either be Xerr, Yerr or Zerr.

NOTE

The error at X = Y = Z = 0 must be 0. This value in the file corresponds to the HomePreset positions in the XY group reference. A terminator (#) must be added at end of each table.

To activate XYZ mapping, the mapping files must be in the ..\admin\config directory of the XPS controller and the following settings must be configured in the system.ini:

- **XMappingFileName:** Name of the mapping file.
- **XMappingXLineNumber:** Total number of lines of each table including the header.
- **XMappingYColumnNumber:** Total number of columns.
- XMappingZDimNumber: Number of tables.
- **XMappingMaxPositionError:** Maximum absolute error in that file must be larger than any entry in the mapping file.
- **YMappingFileName:** Name of the mapping file.
- **YMappingXLineNumber:** Total number of lines of each table including header.
- **YMappingYColumnNumber:** Total number of columns.

- YMappingZDimNumber: Number of tables.
- **YMappingMaxPositionError:** Maximum absolute error in that file must be larger than any entry in the mapping file.
- **ZMappingFileName:** Name of the mapping file.
- **ZMappingXLineNumber:** Total number of lines of each table including header.
- **ZMappingYColumnNumber:** Total number of columns.
- ZMappingZDimNumber: Number of tables.
- **ZMappingMaxPositionError:** Maximum absolute error in that file must be larger than any entry in the mapping file.

The X(Y,Z)MappingXLineNumber, X(Y,Z)MappingYColumnNumber, X(Y,Z)MappingZDimNumber and X(Y,Z)MappingMaxPositionError are only used to check for the correctness of the mapping file.

Example

The following example shows the X error mapping files for an XYZ mapping. Note that it is not necessary to repeat the XY coordinates in the table, Z = -1 to the other tables, Z = 0 and Z = 1.

Matrix of X errors: XYZMapping_X.txt

```
 -1.00     -3.00     -2.00     -1.00      0.00      1.00      2.00      3.00
 -3.00   -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
 -2.00    0.00453  -0.00322   0.00376  -0.00412  -0.00258  -0.00111  -0.00287
 -1.00   -0.00331   0.00445  -0.00769  -0.00126  -0.00153   0.00298   0.00487
  0.00   -0.00787   0.00228  -0.00787   0.00320   0.00154  -0.00169  -0.00369
  1.00    0.00232   0.00210  -0.00342   0.00169   0.00265   0.00169   0.00125
  2.00   -0.00134   0.00308   0.00275  -0.00369   0.00337  -0.00214  -0.00456
  3.00    0.00189  -0.00148   0.00234   0.00458  -0.00333   0.00152   0.00335
#
  0         0         0         0         0         0         0         0
  0       -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
  0        0.00453  -0.00322   0.00376  -0.00412  -0.00258  -0.00111  -0.00287
  0       -0.00331   0.00445  -0.00769  -0.00126  -0.00153   0.00298   0.00487
  0       -0.00787   0.00228  -0.00787   0         0.00154  -0.00169  -0.00369
  0        0.00232   0.00210  -0.00342   0.00169   0.00265   0.00169   0.00125
  0       -0.00134   0.00308   0.00275  -0.00369   0.00337  -0.00214  -0.00456
  0        0.00189  -0.00148   0.00234   0.00458  -0.00333   0.00152   0.00335
#
  1.00      0         0         0         0         0         0         0
  0       -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
  0        0.00453  -0.00322   0.00376  -0.00412  -0.00258  -0.00111  -0.00287
  0       -0.00331   0.00445  -0.00769  -0.00126  -0.00153   0.00298   0.00487
  0       -0.00787   0.00228  -0.00787   0.00320   0.00154  -0.00169  -0.00369
  0        0.00232   0.00210  -0.00342   0.00169   0.00265   0.00169   0.00125
  0       -0.00134   0.00308   0.00275  -0.00369   0.00337  -0.00214  -0.00456
  0        0.00189  -0.00148   0.00234   0.00458  -0.00333   0.00152   0.00335
#
```

Matrix of Y errors: XYZMapping_Y.txt

```
 -1.00     -3.00     -2.00     -1.00      0.00      1.00      2.00      3.00
 -3.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
 -2.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
 -1.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
  0.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
  1.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
  2.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
  3.00  -0.00190  -0.00530   0.00190   0.00125  -0.00190   0.00530   0.00190
#
     0         0         0         0         0         0         0         0
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254         0  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
#
  1.00         0         0         0         0         0         0         0
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
     0  -0.00192  -0.00534   0.00254   0.00125  -0.00137   0.00110   0.00123
#
```

Matrix of Z errors: XYZMapping_Z.txt

```
 -1.00     -3.00     -2.00     -1.00      0.00      1.00      2.00      3.00
 -3.00   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
 -2.00   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
 -1.00   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
  0.00   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
  1.00   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
  2.00   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
  3.00   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
#
     0         0         0         0         0         0         0         0
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
     0   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
     0   -0.0003   -0.0003    0.0003         0   -0.0003   -0.0003    0.0003
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
     0   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
#
  1.00         0         0         0         0         0         0         0
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
     0   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
     0   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
     0   -0.0003   -0.0003    0.0003    0.0003   -0.0003   -0.0003    0.0003
     0   -0.0002   -0.0002    0.0002    0.0002   -0.0002   -0.0002    0.0002
#
```

Verify in the corresponding sections of the stages.ini:

For the X axis:

```
;--- Travels
MinimumTargetPosition =-3      ; unit
HomePreset =0; unit
MaximumTargetPosition =3       ; unit
```

NOTE

The limit travels must be equal or within the X limit positions of the mapping files (shown here +3 and -3).

For the Y axis:

```
;--- Travels
MinimumTargetPosition =-3      ; unit
HomePreset =0; unit
MaximumTargetPosition =3       ; unit
```

NOTE

The limit travels must be equal or within the Y limit positions of the mapping files (shown here +3 and -3).

For Z axis:

```
;--- Travels
MinimumTargetPosition =-1      ; unit
HomePreset =0; unit
MaximumTargetPosition =1       ; unit
```

NOTE

The limit travels must be equal or within the Z limit positions of the mapping files (shown here +1 and -1).

Represents the errors in the X axis.

Represents the errors in the Y axis.

Represents the errors in the Z axis.

```
In the system.ini file:

;--- Mapping XYZ
XMappingFileName = XYZMapping_X.txt
XMappingXLineNumber = 7
XMappingYColumnNumber = 7
XMappingZDimNumber = 3
XMappingMaxPositionError = 0.00787
YMappingFileName = XYZMapping_Y.txt
YMappingXLineNumber = 7
YMappingYColumnNumber = 7
YMappingZDimNumber = 3
YMappingMaxPositionError = 0.00534
ZMappingFileName = XYZMapping_Z.txt
ZMappingXLineNumber = 7
```

ZMappingYColumnNumber = 7
ZMappingZDimNumber = 3
ZMappingMaxPositionError = 0.0003

Use of the functions:

- GroupInitialize(XYZ)
- GroupHomeSearch(XYZ)
- GroupMoveAbsolute(XYZ, 3, 1, 1)

The mapping files must at least cover the minimum and the maximum travel of the XYZ group (they must cover the MinimumTargetPosition and the MaximumTargetPosition for the X, Y and Z positioners, parameters defined in the stages.ini, see section Travels). So in the above example the travel of the X and Y positioners can not be larger than ±3 units, and the travel for the Z positioner can not be larger than ±1 unit. But the travel can be smaller than these. The unit of the data is the same as defined by EncoderResolution in the stages.ini. The data reads as follows: at position (X,Y,Z) = (3.00, 2.00, 1.00), the corrected X position is 2.99848 units (3.00 - 0.00152), the corrected Y position is 2.9989 units (3.00 - 0.00110) and the corrected Z position is 3.0002 units (3.00 + 0.0002). Between two datas, the XPS controller does a linear interpolation of the error. The three mapping files for X, Y, and Z don't need to contain the same X, Y and Z positions.

NOTE

Mapping is a function implemented in the XPS controller to correct errors. But when mapping is activated, it is transparent to the user. At position (X,Y,Z) = (3.00, 1.00, 1.00), the function GroupPositionCurrentGet(XYZ.X) doesn't return 3.00333 (3.00 + 0.00333) but returns 3.00.

7.0 Event Triggers

XPS event triggers work similar to IF/THEN statements in programming. “If” the **event** occurs, “then” an **action** is triggered. Programmers can trigger any action (from a list of possible actions, see section 7.2) at any event (from a large list of possible events, see section 7.1). It is also possible to trigger several actions with the same event. Furthermore, it is possible to link several events to an event configuration. In this case, all events must happen at the same time to trigger the action(s). It is comparable to a logic AND between the different events.

Some events are one-time events like “motion start”. They will trigger an action only once when the event occurs. Some other events have a duration like “motion state”. They will trigger the same action each time (as applicable) as long as the event occurs. For events with duration, the event can be also considered as a statement that is checked whether it is true or not. A third event category are the permanent events “Always” (always happens) and “Timer” (happens every nth servo cycle). They will trigger the action always on every nth servo cycle.

As the XPS controller provides the utmost flexibility in programming event triggers, the user must be careful and consider possible unwanted effects. Some events might have a duration although only one single action is asked. Some other events might never occur. This is especially true when linking several events to an event configuration. The different possible effects are illustrated in section 7.3 by a few examples.

To trigger an action with an event, the event and the associated action must first be configured using the functions **EventExtendedConfigurationTriggerSet**() and **EventExtendedConfigurationActionSet**(). Then, the event trigger is activated using the function **EventExtendedStart**(). When activated, the XPS controller checks for the event at each servo cycle (or each profiler cycle for those events that are motion related) and triggers the action when the event happens. Hence, the maximum latency between the event and the action is equal to the servo cycle (default value 125 µs) or equal to the profiler cycle time (default value 500 µs). For events with duration, it means that the same action is triggered at each servo cycle, i.e. every 125 µs, or at each profiler cycle, i.e. every 500 µs, as long as the event is happening.

Event triggers (and their associated actions) are automatically removed after the event configuration has happened at least once and is no longer true. The only exception is if the event configuration contains any of the permanent events “Always” or “Timer”. In this case the event trigger will always stay active. With the function **EventExtendedRemove**(), any event trigger can get removed.

The function **EventExtendedWait**() can be used to halt a process. It essentially blocks the socket until the event occurs. Once the event occurs, it is deleted. It requires a preceding function **EventExtendedConfigurationTriggerSet**() to define the event at which the process continues.

The functions **EventExtendedGet**() and **EventExtendedAllGet**() return details of the event and action configurations.

7.1 Events

General events are defined as “**Always**”, “**Immediate**” and “**Timer**”. With the event “Always”, an action is triggered each servo cycle, meaning every 125 µs for default value. For events that are defined as “Immediate”, an action is triggered once immediately (during the very next servo cycle). For the events defined as “Timer”, an action is triggered immediately and every nth servo cycle. Here, “n” corresponds to the “FrequencyTicks” defined in the function **TimerSet**(). There are five different timers available that can be selected by the actor (1…5) (Actor is the object that actions/events are linked to).

All events that are motion related (from MotionStart to TrajectoryPulseOutputState in the below table, except MotionDone) refer to the motion profiler of the XPS controller. Thus, events triggered by the motion profiler have a resolution of 500 µs. Consequently, events with duration, such as MotionState, will trigger an action every 500 µs. All motion related events, except MotionDone, have a category such as “Sgamma” or “Jog”. This category refers to the motion profiler. Here, SGamma refers to the profiler used with the function GroupMoveRelative and GroupMoveAbsolute and Jog refers to the profiler used in the Jogging state. The other event categories refer to trajectories. The separator between the category, the actor, and the event name is a dot (.).

NOTE

The table is not an exhaustive list. Refer to the Programmer's Manual for more information.

[Actor.]				[Category.]						Event Name	Parameters			
Group	Positioner	GPIO	TimerX	SGamma	Jog	XYLineArc	Spline	PVT	PT		1	2	3	4
										Immediate	0	0	0	0
										Always	0	0	0	0
			■							Timer	0	0	0	0
	■			■	■					MotionStart	0	0	0	0
	■			■	■					MotionEnd	0	0	0	0
	■			■	■					MotionState	0	0	0	0
	■			■						MotionDone	0	0	0	0
	■			■	■					ConstantVelocityStart	0	0	0	0
	■			■						ConstantVelocityEnd	0	0	0	0
	■			■	■					ConstantVelocityState	0	0	0	0
	■			■						ConstantAccelerationStart	0	0	0	0
	■			■						ConstantAccelerationEnd	0	0	0	0
	■			■						ConstantAccelerationState	0	0	0	0
	■			■						ConstantDecelerationStart	0	0	0	0
	■			■						ConstantDecelerationEnd	0	0	0	0
	■			■						ConstantDecelerationState	0	0	0	0
■						■	■	■	■	TrajectoryStart	0	0	0	0
■						■	■	■	■	TrajectoryEnd	0	0	0	0
■						■	■	■	■	TrajectoryState	0	0	0	0
■						■	■	■	■	ElementNumberStart	Element index	0	0	0
■						■	■	■	■	ElementNumberState	Element index	0	0	0
■						■	■	■	■	TrajectoryPulse	0	0	0	0
■						■	■	■	■	TrajectoryPulseState	0	0	0	0
		■								DILowState	Bit index	0	0	0
		■								DIHighState	Bit index	0	0	0
		■								DILowHigh	Bit index	0	0	0
		■								DIHighLow	Bit index	0	0	0
		■								DIToggled	Bit index	0	0	0
		■								ADCHighLimit	Value	0	0	0
		■								ADCLowLimit	Value	0	0	0
		■								ADCInWindow	min	max	0	0
		■								ADCOutWindow	min	max	0	0
	■									PositionerError	Mask	0	0	0
	■									PositionerHardwareStatus	Mask	0	0	0
	■									ExcitationSignalStart	0	0	0	0
	■									ExcitationSignalEnd	0	0	0	0

	■									WarningFollowingError	0	0	0	0
	■									WaitForPositionLeftToRight	Target position	0	0	0
	■									WaitForPositionRightToLeft	Target position	0	0	0
	■									WaitForPositionToggled	Target position	0	0	0
										DoubleGlobalArrayEqual	Global variable number	value	0	0
										DoubleGlobalArrayDifferent	Global variable number	value	0	0
										DoubleGlobalArrayInferiorOrEqual	Global variable number	value	0	0
										DoubleGlobalArraySuperiorOrEqual	Global variable number	value	0	0
										DoubleGlobalArrayInferior	Global variable number	value	0	0
										DoubleGlobalArraySuperior	Global variable number	value	0	0
										DoubleGlobalArrayInWindow	Global variable number	min	max	0
										DoubleGlobalArrayOutWindow	Global variable number	min	max	0

An event is entirely composed of:

[Actor].[Category].Event Name, Parameter1, Parameter2, Parameter3, Parameter4

Not all event names have a preceding actor and category, but all events have four parameters, even though some parameters are not needed. For these parameters, it is still required to use zero (0) as default.

To define an Event, use the function EventExtendedConfigurationTriggerSet().

Examples

EventExtendedConfigurationTriggerSet (MyGroup.MyPositioner.SGamma.MotionStart, 0, 0, 0, 0)

In this case, the actor is a positioner (MyGroup.MyPositioner) and the event has a category. The event happens when the next motion with the SGamma profiler on the positioner MyGroup.MyPositioner starts. After the motion has started, the event is removed.

EventExtendedConfigurationTriggerSet (MyGroup.XYLineArc.ElementNumberStart, 5, 0, 0, 0)

In this case, the actor is a group (MyGroup) and the event has a category. The event happens when the trajectory element number 5 on the next LineArc trajectory on this group starts.

EventExtendedConfigurationTriggerSet (GPIO4.ADC2.ADCHighLimit, 3, 0, 0, 0)

In this case, the actor is a GPIO name (GPIO4.ADC2) and the event has no category. The event happens when the voltage on the GPIO.ADC2 exceeds 3 Volts.

It is also possible to link different events to an event configuration. The same function EventExtendedConfigurationTriggerSet() is used, and the different events are just separated by a comma. The event combination happens when all individual events happen at the same time. It is comparable to a logic AND between the different events.

Examples

EventExtendedConfigurationTriggerSet (GPIO4.ADC2.ADCHighLimit, 3, 0, 0, 0, MyGroup.MyPositioner.SGamma.MotionState, 0, 0, 0, 0)

This event will happen when the voltage of the GPIO.ADC2 exceeds 3 Volts during a SGamma motion of the MyGroup.MyPositioner.

EventExtendedConfigurationTriggerSet (Always, 0, 0, 0, 0, MyGroup.MyPositioner.SGamma.MotionStart, 0, 0, 0, 0)

This event will happen during each SGamma motion starts of the positioner MyGroup.MyPositioner. The addition of the event Always has the effect of keeping the event after the next motion has been started (see differences compared to the first example above).

The exact meaning of the different events and event parameters are as follows:

Always: Triggers an action ALWAYS, means at each servo cycle.

Event parameter 1 to 4 = 0 by default.

NOTE: This event is PERMANENT until the next reboot. Call the EventExtendedRemove function to remove it.

Immediate: Triggers an action IMMEDIATELY, meaning once during the very next servo cycle:

Event parameter 1 to 4 = 0 by default.

NOTE: This event is TEMPORARY.

Timer: Triggers an action every nth servo cycle, where n is defined with the function TimerSet.

Event parameter 1 to 4 = 0 by default.

NOTE: This event is PERMANENT until the next reboot. Call the EventExtendedRemove function to remove it.

MotionDone: Triggers an action when a position is reached.

Event parameter 1 to 4 = 0 by default.

For the exact definition of MotionDone, please refer to section 3.5.

ConstantVelocityStart: Triggers an action when constant velocity is reached. Event parameter 1 to 4 = 0 by default.

ConstantVelocityEnd: Triggers an action when constant velocity is finished. Event parameter 1 to 4 = 0 by default.

ConstantVelocityState: Triggers an action during constant velocity. Event parameter 1 to 4 = 0 by default.

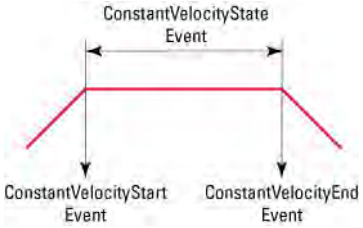


Figure 21: Constant velocity event.

ConstantAccelerationStart: Triggers an action when constant acceleration is reached. Event parameter 1 to 4 = 0 by default.

ConstantAccelerationEnd: Triggers an action when constant acceleration is finished. Event parameter 1 to 4 = 0 by default.

ConstantAccelerationState: Triggers an action during constant acceleration. Event parameter 1 to 4 = 0 by default.

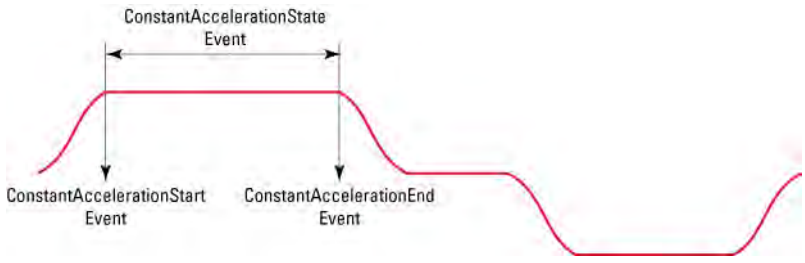


Figure 22: Constant acceleration event.

The same definition applies to ConstantDecelerationStart, ConstantDecelerationEnd and ConstantDecelerationState.

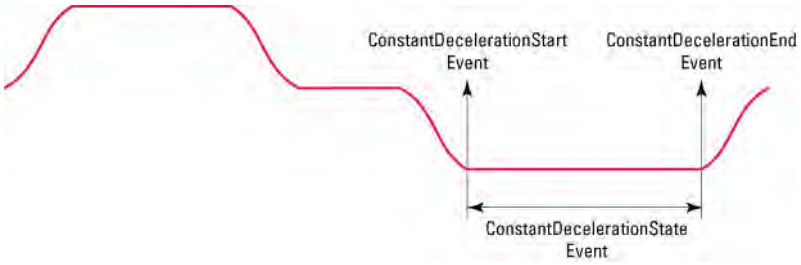


Figure 23: Constant deceleration event.

MotionStart: Triggers an action when motion starts. Event parameter 1 to 4 = 0 by default.

MotionEnd: Trigger an action when motion is ended. Event parameter 1 to 4 = 0 by default. Note, MotionEnd refers to the end of the theoretical motion which is not the same as the definition of MotionDone (see section 3.5).

MotionState: Triggers an action during motion. Event parameter 1 to 4 = 0 by default.

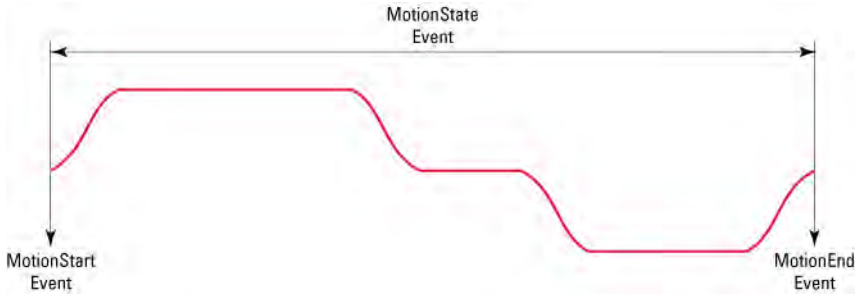


Figure 24: Motion event.

There are also several trajectory events that can be defined:

TrajectoryStart: Triggers an action when the trajectory has started. Event parameter 1 to 4 = 0 by default.

TrajectoryEnd: Triggers an action when the trajectory has stopped. Event parameter 1 to 4 = 0 by default.

TrajectoryState: Triggers an action during trajectory execution. Event parameter 1 to 4 = 0 by default.

Figure 25: Trajectory event.

ElementNumberStart: Triggers an action when the trajectory element number has started. The first event parameter specifies the number of the trajectory element. The other event parameters are 0 by default.

ElementNumberState: Triggers an action during the execution of that trajectory element number. The first event parameter specifies the number of the trajectory element. The other event parameters are 0 by default.

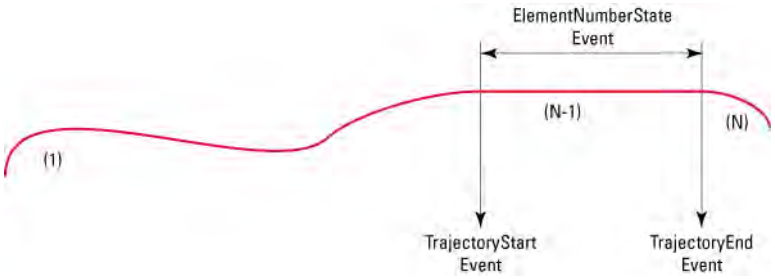


Figure 26: Element number event.

TrajectoryPulse: Triggers an action when a pulse on the trajectory is generated (see chapter 9.0: "Output Triggers" for details). All event parameters are 0 by default.

TrajectoryPulseState: Triggers an action during the trajectory pulse output state, meaning between the start and the end of the trajectory output pulses (see chapter 9.0: "Output Triggers" for details). All event parameters are 0 by default.

ILowState: Triggers an action when the digital input bit is in a low state. The first event parameter is the bit index (0 to 15). The other event parameters are 0 by default.

DILowHigh: Triggers an action when the digital input bit switches from a low state to a high state. The first event parameter is the bit index (0 to 15). The other event parameters are 0 by default.

DIHighState: Triggers an action when the digital input bit is in a high state. The first event parameter is the bit index (0 to 15). The other event parameters are 0 by default.

DIHighLow: Triggers an action when the digital input bit switches from a high to a low state. The first event parameter is the bit index (0 to 15). The other event parameters are 0 by default.

DIToggled: Triggers an action when the digital input bit switches from low to high or from high to low. The first event parameter is the bit index (0 to 15). The other event parameters are 0 by default.

ADCHighLimit: Triggers an action when the analog input value exceeds the limit. The first event parameter is the limit value in volts. The other event parameters are 0 by default.

ADCLowLimit: Triggers an action when the analog input value is below the limit. The first event parameter is the limit value in volts. The other event parameters are 0 by default.

ADCInWindow: Triggers an action when the analog input value is inside the window defined by min and max values. The first event parameter is the minimum value in volts and the second event parameter is the maximum value in volts. The other event parameters are 0 by default.

ADCOutWindow: Triggers an action when the analog input value is out of the window defined by min and max values. The first event parameter is the minimum value in volts and the second event parameter is the maximum value in volts. The other event parameters are 0 by default.

PositionerError: Triggers an action when the current positioner error applied with the error mask (for the 32 bit register) results in a value other than zero. The first event parameter specifies

the error mask in a decimal format. The other event parameters are 0 by default.

NOTE

Refer to the Programmer's Manual for an error mask table.

Examples

EventExtendedConfigurationTriggerSet (MyGroup.MyPositioner.PositionerError, 2, 0, 0, 0)

This event happens when the positioner MyGroup.MyPositioner has a fatal following error.

EventExtendedConfigurationTriggerSet (MyGroup.MyPositioner.PositionerError, 12, 0, 0, 0)

This event happens when the positioner MyGroup.MyPositioner has either a home search time out or a motion done time out.

PositionerHardwareStatus: Triggers an action when the current hardware status applied with the error mask results in a value other than zero. The first event parameter specifies the status mask in decimal format. The other event parameters are 0 by default.

NOTE

For more information regarding PositionerHardwareStatus refer to the Programmer's Manual.

Example

EventExtendedConfigurationTriggerSet (MyGroup.MyPositioner.PositionerHardwareStatus, 768, 0, 0, 0)

This event happens when the positioner MyGroup.MyPositioner either the plus end of run or a minus end of run is detected.

WarningFollowingError: Triggers an action when the following error exceeds the warning following error value. In the PositionCompare mode (activated by the PositionerPositionCompareEnable() function), during a move (relative or absolute) and inside the zone set by PositionerPositionCompareSet(), if the current following error exceeds the WarningFollowingError value, the PositionCompareWarningFollowingErrorFlag is activated and the move returns a corresponding error (-120 : Warning following error during move with position compare enabled).

To reset the PositionCompareWarningFollowingErrorFlag, send the PositionerPositionCompareDisable() function.

The WarningFollowingError is set to FatalFollowingError (defined in stages.ini file) by default, but it can be modified with PositionerWarningErrorSet().

Example

EventExtendedConfigurationTriggerSet

(MyGroup.MyPositioner. WarningFollowingError, 0, 0, 0, 0)

This event happens when the positioner MyGroup.MyPositioner has a following error that exceeds the warning following error value.

DoubleGlobalArrayEqual: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is equal to the value to check. The variable can be modified by using the DoubleGlobalArraySet() function.

DoubleGlobalArrayDifferent: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is different from the value to check. The variable can be modified by using the DoubleGlobalArraySet() function.

DoubleGlobalArrayInferiorOrEqual: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is less than or equal to the value to check. The variable can be modified by using the DoubleGlobalArraySet() function.

DoubleGlobalArraySuperiorOrEqual: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is greater than or equal to the value to check. The variable can be modified by using the DoubleGlobalArraySet() function.

DoubleGlobalArrayInferior: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is lower than the value to check. The variable can be modified by using the DoubleGlobalArraySet() function.

DoubleGlobalArraySuperior: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is higher than the value to check. The variable can be modified by using the DoubleGlobalArraySet() function.

DoubleGlobalArrayInWindow: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is superior to MinValue and inferior to MaxValue.

DoubleGlobalArrayOutWindow: Triggers an action when the value of the variable in the DoubleGlobalArray and referenced by the global variable number is outside the interval defined by MinValue and MaxValue.

WaitForPositionLeftToRight:

Triggers an action when the target position is detected during a displacement or if the target position has already passed, the action is triggered at the beginning of the displacement. The target position is checked only for all **positive displacements**. The first event parameter is the target position. The other event parameters are 0 by default.

NOTE

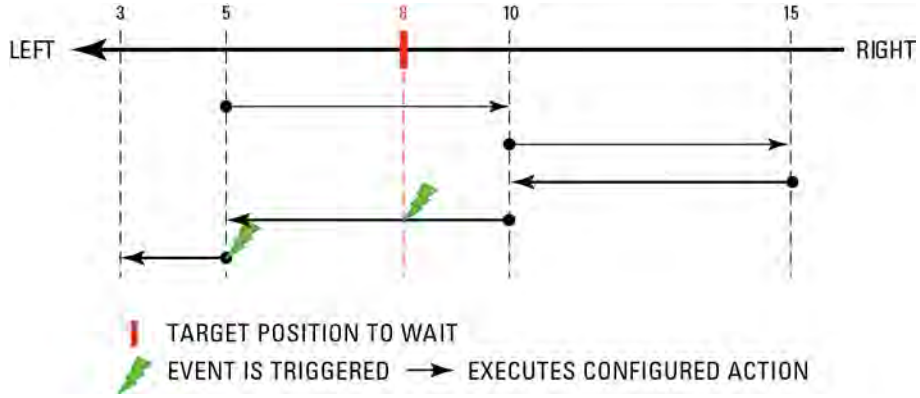
It is recommended to use this event trigger only after the home search is done. Once the event has occurred, it is deleted automatically. Reactivate the event trigger with the API function EventExtendedStart().

WaitForPositionRightToLeft:

Triggers an action when the target position is detected during a displacement or if the target position has already passed, the action is triggered at the beginning of the displacement. The target position is checked only for all **negative displacements**. The first event parameter is the target position. The other event parameters are 0 by default.

NOTE

It is recommended to use this event trigger only after the home search is done. Once the event has occurred, it is deleted automatically. Reactivate the event trigger with the API function EventExtendedStart().



WaitForPositionToggled:

Triggers an action when the target position is detected during a displacement or if the target position has already passed, the action is triggered at the beginning of the displacement. The target position is checked for **all displacements**. The first event parameter is the target position. The other event parameters are 0 by default.

NOTE

It is recommended to use this event trigger only after the home search is done. Once the event has occurred, it is deleted automatically. Reactivate the event trigger with the API function EventExtendedStart().

Example

GroupInitialize(MyGroup)

GroupHomeSearch(MyGroup)

EventExtendedConfigurationTriggerSet(MyPositioner.**WaitForPositionToggled**, 8.0, 0, 0, 0)

EventExtendedConfigurationActionSet(GatheringRun, 10000, 1, 0, 0)

EventExtendedStart(int *ID)

The event will trigger the configured action, Gathering Run, when the target position is detected for MyPositioner or at the beginning of the displacement if the target position is already passed.

7.2 Actions

There are several actions that can be triggered by the events discussed previously. Users have the full flexibility to trigger any action (out of the list of possible actions) at any event (out of the list of possible events). It is also possible to trigger several actions at the same event by adding several sets of parameters to the function **EventExtendedConfigurationActionSet**(), similar to how it is done with events.

NOTE

The table is not an exhaustive list. Refer to the Programmer's Manual for more information.

[Actor.]				Action Name	Parameters			
Group	Positioner	GPIO	TimerX		1	2	3	4
		■		DACSet.CurrentPosition	Positioner name	Gain	Offset	0
		■		DACSet. CurrentVelocity	Positioner name	Gain	Offset	0
		■		DACSet. SetpointPosition	Positioner name	Gain	Offset	0
		■		DACSet. SetpointVelocity	Positioner name	Gain	Offset	0
		■		DACSet. SetpointAcceleration	Positioner name	Gain	Offset	0
		■		DACSet.Value	Value	0	0	0
		■		DOPulse	Mask	0	0	0
		■		DOToggle	Mask	0	0	0
		■		DOSet	Mask	Value	0	0
				EventRemove	Trigger identifier *(-1 for itself)*	0	0	0
				ExecuteCommand	Function name	Parameters *(Between {} and separator is the semi-column)*	Task name	
				ExecuteTCLScript	TCL file name	Task name	Arguments	
				ExternalGatheringRun	Nb of points	1	0	0
				GatheringOneData	0	0	0	0
				GatheringRun	Nb of points	Divisor	0	0
				GatheringRunAppend	0	0	0	0
				GatheringStop	0	0	0	0
				GlobalArrayDoubleSet	Global variable number	Double value	0	0
				GlobalArrayStringSet	Global variable number	String value	0	0
				KillTCLScript	Task name	0	0	0
■				MoveAbort	0	0	0	0
■				MoveAbortFast	Deceleration multiplier	0	0	0
				SynchronizeProfiler	0	0	0	0

CAUTION

Certain events like MotionState have a duration. These events trigger the associated action in each motion profiler cycle as long as the event is true. For example, associating the action DOToggle with the event MotionState will toggle the value of the digital output in each profiler cycle as long as the MotionState event is true.

An event doesn't reset the action after the event: For example, to set a digital output to a certain value during a constant velocity state and to set it to its previous value afterwards, two event triggers are needed: One to set to the digital output of the desired value at the event ConstantVelocityStart and another one to set it to its original value at the event ConstantVelocityEnd. The same effect CANNOT be achieved by using the event ConstantVelocityState by itself.

An action is composed entirely of:

[Actor].ActionName, Parameter1, Parameter2, Parameter3, Parameter4

Not all action names have a preceding actor, but all actions have four parameters. Even though all four parameters may not be defined in an action, it is still required to have an entry, with zero (0) as the default.

To define an action, use the function **EventExtendedConfigurationActionSet**().

Example:

EventExtendedConfigurationActionSet(GPIO3.DO.DOToggled,4,0,0,0)

In this case the actor is the digital output GPIO3.DO and the action is to toggle the output. The value 4 refers to bit #3, 00000100. Hence, this action toggles the value of bit 3 on the digital output GPIO.DO.

EventExtendedConfigurationActionSet(ExecuteTCLScript,Example.tcl,1,0,0)

The action ExecuteTCLScript has no preceding actor. This action will execute the TCL script "Example.tcl". The task name is 1 and the TCL script has no arguments (a zero for the third parameter means there are no arguments).

EventExtendedConfigurationActionSet(GatheringRun,1000,8,0,0)

The action GatheringRun has no preceding actor. This action will start an internal data gathering. It will gather a total of 1000 data points, one data point every 8th servo cycle.

It is also possible to trigger several actions with the same event. To do so, just define another action in the SAME function. Several actions must be separated by a comma (,).

Example:

EventExtendedConfigurationTriggerSet (MyGroup.MyPositioner.PositionerError, 2, 0, 0, 0)

EventExtendedConfigurationActionSet (ExecuteTCLScript, ShutDown.tcl, 1, 0, 0, ExecuteTCLScript, ErrorDiagnostic.tcl, 2, 0, 0)

EventExtendedStart()

In this example, the TCL scripts ShutDown.tcl and ErrorDiagnostic.tcl are executed when a fatal following error is detected on the positioner MyGroup.MyPositioner.

The exact meaning of the different actions and action parameters is as follows:

DOToggle: This action is used to reverse the value of one or many bits of the Digital Output. When using this action with an event that has some duration (for example motion state) the value of the bits will be toggled at each profiler cycle as long as the event occurs.

Action Parameter #1 – Mask	The mask defines which bits on the GPIO output will be toggled (change their value). For example, if the GPIO output is an 8 bit

	output and the mask is set to 4 then the equivalent binary number is 00000100. So as an action, bit #3 will be toggled.
Action Parameter #2 to #4	These parameters are 0 by default.

DOPulse: This action is used to generate a positive pulse on the Digital Output. The duration of the pulse is 1 microsecond. To function, the bits on which the pulse is generated should be set to zero before. When using this action with an event that has some duration (for example motion state), a 1 µs pulse will be generated at each cycle of the motion profiler (or every 500 µs for default value) as long as the event occurs.

Action Parameter #1 – Mask	The mask defines on which bits on the GPIO output the pulse will be generated. For example, if the GPIO output is an 8 bit output and the mask is set to 6 then the equivalent binary number is 00000110. So as an action, a 1 µs pulse will be generated on bit #2 and #3 of the GPIO output.
Action Parameter #2 to #4	These parameters are 0 by default.

DOSet: This action is used to modify the value of bit(s) on a Digital Output.

Action Parameter #1 – Mask	The mask defines which bits on the GPIO output are being addressed. For example, if the GPIO output is an 8 bit output and the mask is set to 26 then the equivalent binary number is 00011010. Therefore with a Mask setting of 26, only bits # 2, #4 and #5 are being addressed on the GPIO output.
Action Parameter #2 – Value	This parameter sets the value of the bits that are being addressed according to the Mask setting. For example since a Mask setting of 26, bits #2, #4 and #5 can be modified, a value of 8 (00001000) will set the bits #2 and #5 to 0 and the bit #4 to 1.
Action parameter #3 and #4	These parameters are 0 by default.

DACSet.CurrentPosition and **DACSet.SetpointPosition:** This action sets a voltage on the Analog output in relation to the actual (current) or theoretical (Setpoint) position. The gain and offset are used to calibrate the output. This action makes the most sense with events that have some duration (always, MotionState, ElementNumberState, etc.) as the analog output will be updated at each servo cycle or at each profiler cycle as long as the event occurs. When used with events that have no duration (like MotionStart or MotionEnd), the analog output is only updated once and this value is kept until it is changed.

Action Parameter #1 – Positioner Name	This parameter defines the name of the positioner on which the position value is used.
Action Parameter #2 – Gain	The position value is multiplied by the gain value. For example, if the gain is set to 10 and the position value is 1 mm (or any other unit), then the output voltage is 10 V.
Action Parameter #3 – Offset	The offset value is used to correct for any voltage that may already be present in the Analog output.

Analog output = Position value * gain + offset

Action parameter #4	This parameter is 0 by default.

DACSet.CurrentVelocity and **DACSet.SetpointVelocity:** This action sets a voltage on the Analog output relative to the actual (current) or theoretical (Setpoint) velocity. The gain and the offset are used to calibrate the output. This action makes most sense with events that have duration (Always, MotionState, ElementNumberState, etc.) as the analog output is updated at each servo cycle or at each profiler cycle as long as the event occurs. When used with events that have no duration (like MotionStart or MotionEnd), the analog output is only updated once and this value is kept until it is changed.

Action Parameter #1 – Positioner Name	This parameter defines the name of the positioner in which the Velocity value is used.
Action Parameter #2 – Gain	The Velocity value is multiplied by the gain value. For example if the gain is set to 10 and the velocity value is 1 mm/s (or any other velocity unit), then the output voltage is 10 V.
Action Parameter #3 – Offset	The offset value is used to correct for any voltage that may initially be present in the Analog output.

Analog output = Velocity value * gain + offset

Action parameter #4	This parameter must be 0 by default.

DACSet.SetpointAcceleration: This action is used to output a voltage on the Analog output to form an image of the theoretical acceleration. The gain and the offset are used to calibrate this image. This action makes most sense with events that have duration (Always, MotionState, ElementNumberState, etc.) as the analog output will be updated at each servo cycle or at each profiler cycle as long as the event lasts. When used with events that have no duration (like MotionStart or MotionEnd), the analog output is only updated once and keep this value until it is changed.

Action Parameter #1 – Positioner Name	This parameter defines the name of the positioner in which the SetpointAcceleration is used to output in the analog output.
Action Parameter #2 – Gain	The SetpointAcceleration is multiplied by the gain value. For example, if the gain is set to 10 and the corrected SetpointAcceleration is 1 mm/s^2 then the output voltage will be 10 V.
Action Parameter #3 – Offset	The offset value is used to correct for any voltage that may initially be present in the Analog output.

Analog output = SetpointAcceleration value * gain + offset

Action parameter #4	This parameter is 0 by default.

NOTE

The gain can be any constant value used to scale the output voltage and the offset value can be any constant value used to correct for any offset voltage in the analog output.

ExecuteTCLScript: This action executes a TCL script on an event.

Action Parameter #1 – TCL File Name — This parameter defines the file name of the TCL program.

Action Parameter #2 – TCL Task Name — Since several TCL scripts can run simultaneously different or even the same, the TCL Task Name is used to track individual TCL programs. For example, the TCL Task Name stops a particular program without stopping all other TCL programs that are running simultaneously.

Action Parameter #3 – TCL Argument List — The Argument list is used to run the TCL scripts with input parameters. For the argument parameter, any input can be used (number, string). These parameters are used inside the script. To get the number of arguments, use $tcl_argc" inside the script. To get each argument, use "$tcl_argc($i)" inside the script. For example, this parameter can be used to specify a number of loops inside the TCL script. A zero (0) for this parameter means there are no input arguments.

Action parameter #4 — This parameter is 0 by default.

KillTCLScript: This action stops a TCL script on an event.

Action parameter #1 – Task name — This parameter defines which TCL script is stopped. Since several TCL scripts can run simultaneously, different or even the same script, the TCL Task Name is used to track individual TCL programs.

Action parameter #2 to #4 — These parameters are 0 by default.

GatheringOneData: This action acquires one data as defined by the function GatheringConfigurationSet. Different from the GatheringRun (see next action), which generates a new gathering file, the GatheringOneData appends the data to the current gathering file stored in memory. In order to store the data in a new file, first launch the function GatheringReset, which deletes the current gathering file from memory.

Action parameter #1 to #4 — These parameters are 0 by default.

GatheringRun: This action starts an internal data gathering. It requires that an internal gathering was previously configured with the function GatheringConfigurationSet. The gathering must be launched by a punctual event and does not work with events that have duration.

Action Parameter #1 – NbPoints — This parameter defines the number of data acquisitions. NbPoints multiplied by the number of gathered data types must be smaller than 1,000,000. For instance, if 4 types of data are collected, NbPoints can not be larger than 250,000 (4*250,000 = 1,000,000).

Action Parameter #2 — This parameter is 1 by default.

Action Parameter #3 and #4 — These parameters are 0 by default.

GatheringRunAppend: This action continues a gathering previously stopped with the action GatheringStop, see next action.

Action parameter #1 to #4 — These parameters are 0 by default.

GatheringStop: This action halts a data gathering previously launched by the action GatheringStart. Use the action GatheringRunAppend to continue data gathering. Note that the action GatheringStop does not automatically store the gathered data from the buffer to the flash disk of the controller. To store data, use the function GatheringStopAndSave. For more details about data gathering, refer to chapter 8.0: "Data Gathering".

Action parameter #1 to #4	These parameters are 0 by default.

ExternalGatheringRun: This action starts an external data gathering. It requires that an external data gathering was previously configured with the function GatheringExternalConfigurationSet. The gathering must be launched by a punctual event and does not work with events that have duration.

Action Parameter #1 – NbPoints	This parameter defines the number of data acquisitions. NbPoints multiplied by the number of gathered data types must be smaller than 1,000,000. For instance, if 4 types of data are collected, NbPoints can not be larger than 250,000 (4*250,000 = 1,000,000).
Action Parameter #2 – Divisor	This parameter defines every Nth number of the trigger input signal at which the gathering will take place. This parameter must be an integer and greater than or equal to 1. For example if the divisor is set to 5 then gathering will take place every 5th trigger on the trigger input signal.
Action Parameter #3 and #4	These parameters are 0 by default.

For further details on data gathering, see chapter 8.0: "Data Gathering".

MoveAbort: This action stops (abort) a motion on an event. It is similar to sending a MoveAbort() function on the event. After stopping, the group is in the READY state.

Action Parameter #1 to #4	These parameters are 0 by default.

7.3 Functions

The following functions are related to event triggers:

- EventExtendedConfigurationTriggerSet(): This function configures one or several events. In the case of several events, the different events are separated by a comma (,) in the argument list. Before activating an event, one or several actions must be configured with the function EventExtendedConfigurationActionSet(). Only then, the event and the associated action(s) can be activated with the function EventExtendedStart().
- EventExtendedConfigurationTriggerGet(): This function returns the event configuration defined by the last EventExtendedConfigurationTriggerSet() function.
- EventExtendedConfigurationActionSet(): This function associates an action to the event defined by the last EventExtendedConfigurationTriggerSet() function.
- EventExtendedConfigurationActionGet(): This function returns the action configuration defined by the last EventExtendedConfigurationActionSet() function.
- EventExtendedStart(): This function launches (activates) the last configured event and the associated action(s) defined by the last EventExtendedConfigurationTriggerSet() and EventExtendedConfigurationActionSet() and returns an event identifier. When activated, the XPS controller checks for the event at each servo cycle (or at each profiler cycle for those events that are motion related) and triggers the action when

the event occurs. Hence, the maximum latency between the event and the action is equal to the servo cycle time (default value 125 µs) or equal to the profiler cycle time (default value 500 µs) for motion related events. For events with duration, it also means that the same action is triggered at each servo cycle, meaning every 125 µs, or at each profiler cycle, which is every 500 µs as long as the event occurs.

Event triggers (and their associated action) are automatically removed after the event configuration has happened at least once and is no longer true. The only exception is if the event configuration contains any of the permanent events “Always” or “Trigger”. In this case the event trigger will always stay active. With the function EventExtendedRemove(), any event trigger can get removed.

- EventExtendedWait(): This function halts a process (essentially by blocking the socket) until the event defined by the last EventExtendedConfigurationTriggerSet() occurs.
- EventExtendedRemove(): This function removes the event trigger associated with the defined event identifier.
- EventExtendedGet(): This function returns the event configuration and the action configuration associated with the defined event identifier.
- EventExtendedAllGet(): This function returns, for all active event triggers, the event identifier, the event configuration and the action configuration. The details of the different event triggers are separated by a comma (,).

7.4 Examples

Below is a table that shows possible events that can be associated with possible actions. Some of these examples however, may have unwanted results. Since the XPS controller provides great flexibility to trigger almost any action at any event, the user must be aware of the possible unwanted effects.

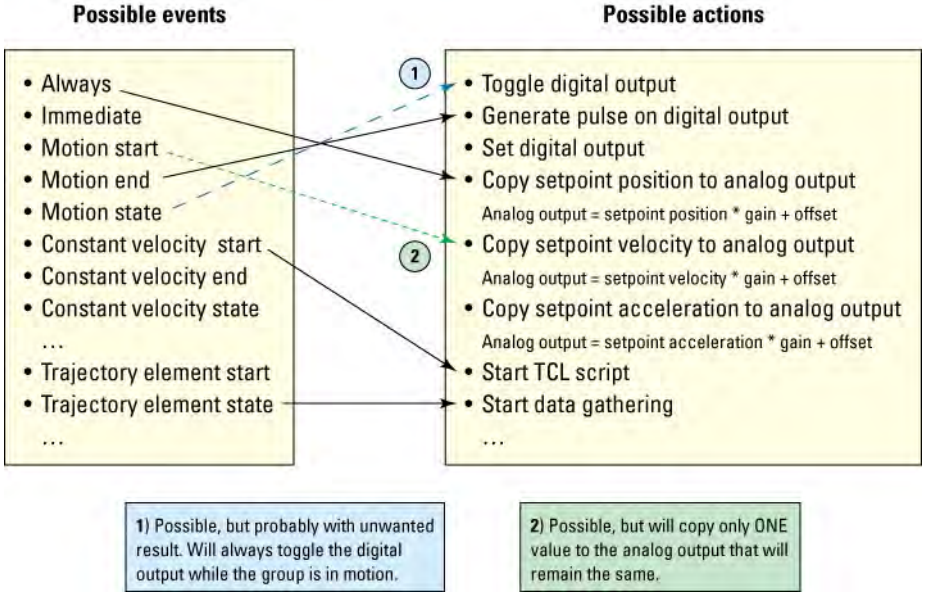


Figure 27: Possible events.

Examples

1. **EventExtendedConfigurationTriggerSet (G1.P1.SGamma.ConstantVelocityStart, 0, 0, 0, 0)**

   EventExtendedConfigurationActionSet (GPIO3.DO.DOSet, 4, 4, 0, 0)

   EventExtendedStart()

   GroupMoveAbsolute (G1.P1, 50)

   In this example, when positioner G1.P1 reaches constant velocity, bit #3 on the digital output on connector number 1 is set to 1 (Note: 4 = 00000100). Note, that the state of the bit will not change when the constant velocity of the positioner has ended. In order to do so, a second event trigger would be required (see next example).

2. **EventExtendedConfigurationTriggerSet (G1.P1.SGamma.ConstantVelocityStart, 0, 0, 0, 0)**

   EventExtendedConfigurationActionSet (GPIO3.DO.DOSet, 4, 4, 0, 0)

   EventExtendedStart()

   EventExtendedConfigurationTriggerSet (G1.P1.SGamma.ConstantVelocityEnd, 0, 0, 0, 0)

   EventExtendedConfigurationActionSet (GPIO3.DO.DOSet, 4, 0, 0, 0)

   EventExtendedStart()

   GroupMoveAbsolute (G1.P1, 50)

   In this example, when positioner G1.P1 reaches constant velocity, bit #3 on the digital output on connector number 1 is set to 1 (Note: 4 = 00000100) and when the constant velocity of the positioner G1.P1 is over, bit #3 will be set to zero. Note, that the same effect can not be reached with the event name ConstantVelocityState. After both events have happened, the event triggers will get automatically removed. In order to trigger the same action at each motion, it is required to link the events with the event “Always” (see next example). This link will avoid that the event trigger gets removed after it is not happening anymore.

3. **EventExtendedConfigurationTriggerSet (Always, 0, 0, 0, 0, G1.P1.SGamma.ConstantVelocityStart, 0, 0, 0, 0)**

   EventExtendedConfigurationActionSet (GPIO3.DO.DOSet, 4, 4, 0, 0)

   EventExtendedStart()

   EventExtendedConfigurationTriggerSet (Always, 0, 0, 0, 0, G1.P1.SGamma.ConstantVelocityEnd, 0, 0, 0, 0)

   EventExtendedConfigurationActionSet (GPIO3.DO.DOSet, 4, 0, 0, 0)

   EventExtendedStart()

   GroupMoveAbsolute (G1.P1, 50)

   GroupMoveAbsolute (G1.P1, -50)

   In this example, when positioner G1.P1 reaches constant velocity, bit #3 on the digital output on connector number 1 is set to 1 (Note: 4 = 00000100) and when the constant velocity of the positioner G1.P1 is over, bit #3 will be set to zero. Different from the previous example, adding the event “Always” avoids the event trigger being removed after the event is over. Hence, the state of the bit #3 will change with every beginning and with every end of the constant velocity state of a motion.

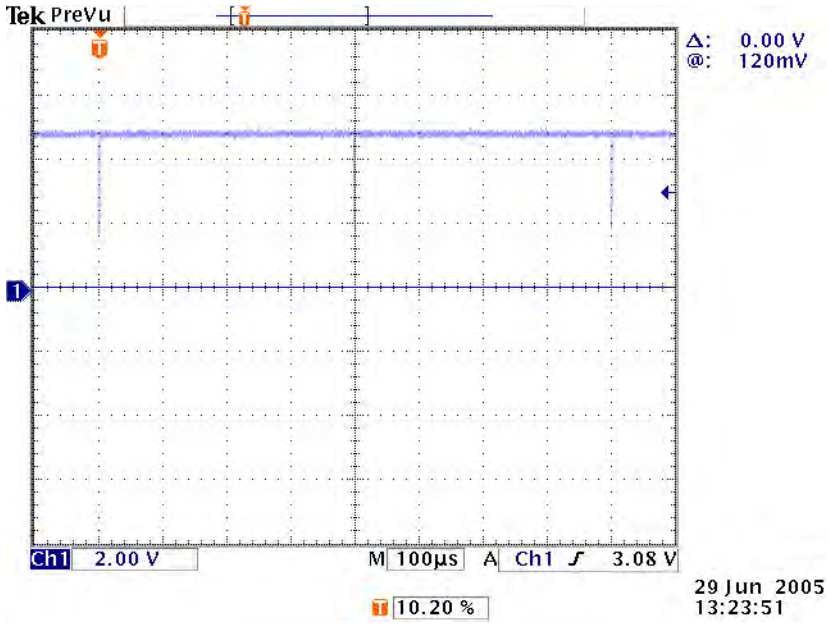
4. **EventExtendedConfigurationTriggerSet (G1.P1.SGamma.ConstantVelocityState, 0, 0, 0, 0)**

   EventExtendedConfigurationActionSet (GPIO3.DO.DOSet, 255, 0, 0, 0)

   EventExtendedStart()

   GroupMoveAbsolute (G1.P1, 50)

   In this example, during the constant velocity state of the positioner G1.P1, 1 µs pulses are generated on all 8 bits in the digital output on connector number 1, at every cycle of the motion profiler (Note: 255 = 11111111). The cycle time of the motion profiler is 500 µs for default value, so pulses are generated every 500 µs (see picture below).



5. **EventExtendedConfigurationTriggerSet (Always, 0, 0, 0, 0)**

   EventExtendedConfigurationActionSet (GPIO4.DAC1.DACSet.SetpointPosition, G1.P1, 0.1, -10, 0, GPIO4.DAC2.DACSet.SetpointVelocity, G1.P1, 0.5, 0, 0)

   EventExtendedStart()

   In this example, the analog output #1 on GPIO4 will always output a voltage in relation to the SetpointPosition of the positioner G1.P1, and the output #2 on GPIO4 will always output a voltage in relation to the SetpointVelocity of the same positioner. The gain on output #1 is set to 0.1 V/unit and the offset to -10 V. This means that when the stage is at the position 0 units, a voltage of -10 V will be sent. When the stage is at the position 10 units, a voltage of -9 V will be sent.
   Here, the event “Always” means that these values will be updated every servo cycle, means every 0.125 ms for the default servo cycle rate. If instead of the event “Always”, the event “Immediate” will be used, only the most recent values will be sent and kept. If instead of the event “Always”, a motion related event such as MotionState is used, the update will only happen at every profiler cycle, or every 0.5 ms for the default profile generator rate.

6. **TimerSet(Timer1,8000)**

   EventExtendedConfigurationTriggerSet (Timer1.Timer, 0, 0, 0, 0)

   EventExtendedConfigurationActionSet (GPIO3.DO.DOToggle, 255, 0, 0, 0)

   EventExtendedStart()

   EventExtendedRemove(1)

   The function Timer() sets the Timer1 at every 8,000th servo cycle. Hence, in this example, the digital output on connector number 1 will be toggled (Note: 255 =

11111111). The event Timer is permanent. In order to remove the event trigger, use the function EventExtendedRemove() with the associated event identifier (1 in this case).

7. **MultipleAxesPVTPulseOutputSet(G1,2,20,1)**

   GatheringConfigurationSet(G1.P1.CurrentPosition)

   EventExtendedConfigurationTriggerSet(Always, 0,0,0,0,G1.PVT.TrajectoryPulse,0,0,0,0)

   EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

   EventExtendedStart()

   MultipleAxesPVTExecution(G1,Traj.trj,1)

   In this example, the generation of an output pulse every one second between the 2nd and the 20th element in the next PVT trajectory executed on the group G1 is first defined (function MultipleAxisPVTPulseOutputSet). Then, data gathering is defined (CurrentPosition of positioner G1.P1).

   Hence, in this example, with every trajectory pulse, one data point is gathered and appended to the current gathering file in memory. Here, adding the event TrajectoryPulse with the permanent event Always makes sure that the event trigger is always active. Without the event Always, only one data point will be gathered.

   This is because any event is automatically removed once it happens and does not happening in the next servo or profiler cycle (which is the case here as a pulse is only generated every one second).

   Please note that the action GatheringOneData appends data to the current data file. In order to store the data in a new file it is required to first launch the function GatheringReset() which deletes the current data file from memory.

8. **GatheringConfigurationSet(G1.P1.CurrentPosition)**

   EventExtendedConfigurationTriggerSet (G1.P1.SGamma.MotionStart,0,0,0,0)

   EventExtendedConfigurationActionSet(GatheringRun,20,1000,0,0)

   EventExtendedStart()

   GroupMoveAbsolute (G1.P1, 50)

   GatheringStopAndSave()

   In this example, an internal data gathering of 20 data points every 0.1 second (every 800th servo cycle) is launched with the start of the next motion of the positioner G1.P1. The type of data that gathered is defined with the function GatheringConfigurationSet (CurrentPosition of positioner G1.P1). To store the data from internal memory to the flash disk in the XPS controller, send the function GatheringStopAndSave(). The GatheringRun deletes the current data file in internal memory (in contrast to the GatheringOneData which appends data to the current file). Also, the function GatheringStopAndSave() stores the data file under a default name Gathering.dat on the flash disk of the XPS controller and will overwrite any older file of the same name in the same folder. Hence, make sure to store valuable data files under a different name before a GatheringStopAndSave().

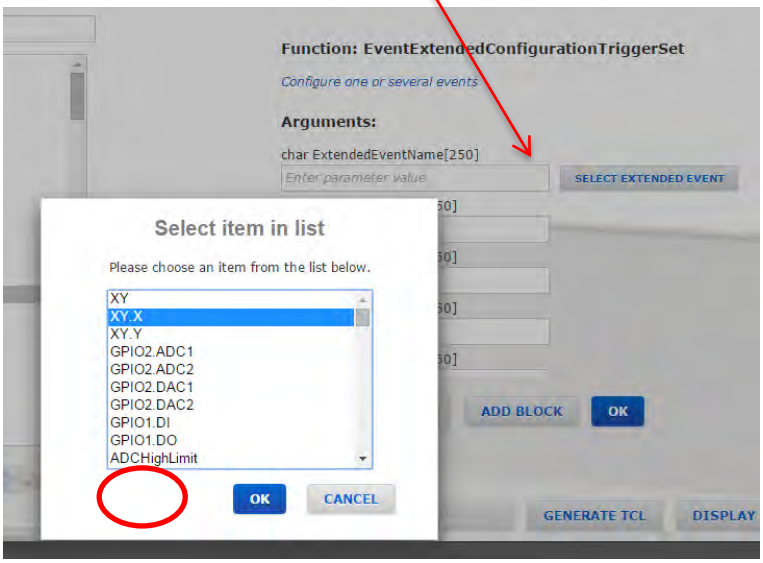
NOTE

When using the function EventExtendedConfigurationTriggerSet() or EventExtendedConfigurationActionSet() from the terminal screen of the XPS utility, the syntax for one parameter is not directly accessible. For instance, for the event XY.X.SGamma.MotionStart, first select XY.X from the choice list. Then, click on the choice field again and select SGammaMotionStart. See also screen shots below.

For specifying more than one data type, use the "ADD BLOCK" button. Select the next parameter as described above.

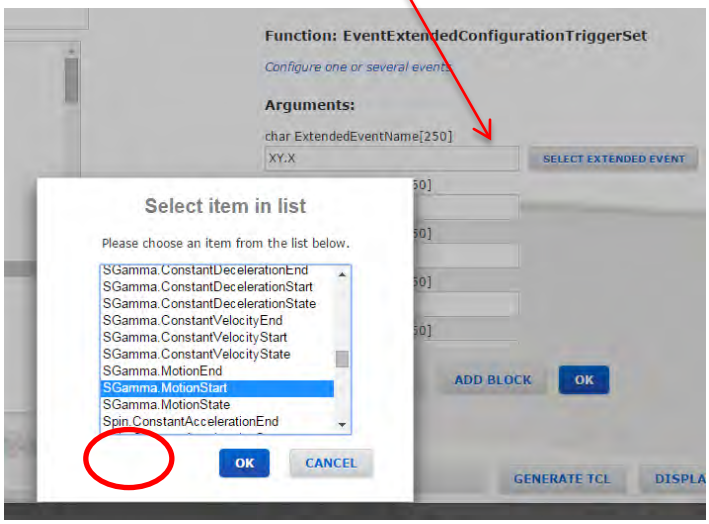
Step 1:

Click **"SELECT EXTENDED EVENT"** then select the positioner name and click **"OK"**.



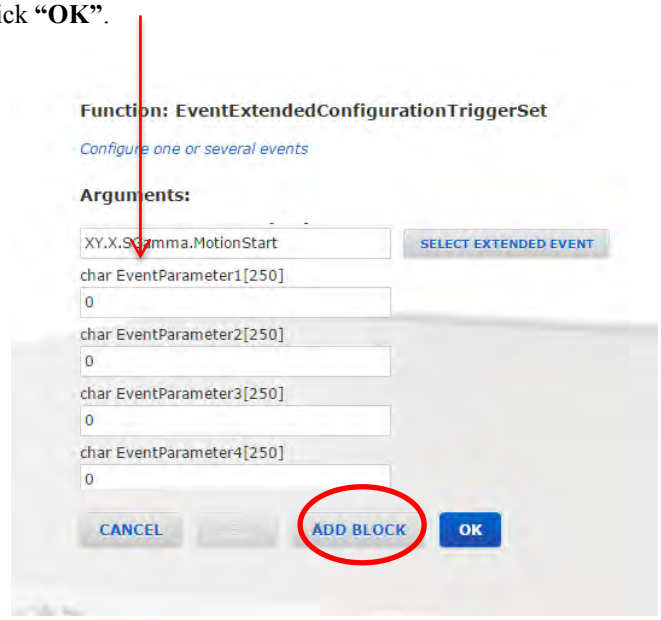
Step 2:

Click **"SELECT EXTENDED EVENT"** again then select the parameter name and click **"OK"**.



Step 3:

Define event parameters. To add another event click **"ADD BLOCK".** Repeat Step 1 and Step 2, or else click **"OK"**.



8.0 Data Gathering

The XPS controller provides four methods for data gathering:

1. Time-based (internal) data gathering. With this method one data set is gathered for every n^{th} servo cycle.
2. Event-based (internal) data gathering. With this method one data set is gathered at an event.
3. Function-based (internal) data gathering. With this method one data set is gathered by a function.
4. Trigger-based (external) data gathering. With this method one data set is gathered for every n^{th} external trigger input (see also chapter 9.0: "Output Triggers ").

Method 1, 2, and 3, these are also referred to as internal or servo cycle synchronous data gathering. With the trigger-based data gathering, this is also referred to as an external data gathering, as the event that triggers the data gathering or the receipt of a trigger input, is asynchronous to the servo cycle.

The time-based, the event-based and the function-based data gathering store the data in a common file called gathering.dat. The trigger-based (external) data gathering stores the data in a different file, called ExternalGathering.dat. The type of data that can be gathered differs also between the internal and the external data gathering.

Before starting any data gathering, the type of data to be gathered needs to be defined using the functions GatheringConfigurationSet() (in case of an internal data gathering) or ExternalGatheringConfigurationSet() (in case of an external data gathering). Refer to the Programmer's Manual and the Gathering functions for a complete list of data types.

During data gathering, new data is appended to a buffer. With the functions GatheringCurrentNumberGet() and GatheringExternalCurrentNumberGet(), the current number of data sets in this buffer and the maximum possible number of data sets that fits into this buffer can be recalled. The maximum possible number of data sets equals 1,000,000 divided by the number of data types belonging to one data set.

The function GatheringDataGet(index) returns one set of data from the buffer. Here, the index 0 refers to the 1st data set, the index (n-1) to the n-th data set. When using this function from the Terminal screen of the XPS utility, the different data types belonging to one data line are separated by a semicolon (;).

To save the data from the buffer to the flash disk of the XPS controller, use the functions GatheringStopAndSave() and GatheringExternalStopAndSave(). These functions will store the gathering files in the XPS controller under the name Gathering.dat (with function GatheringStopAndSave() for internal gathering) or GatheringExternal.dat (with function GatheringExternalStopAndSave() for external gathering). To rename the gathering file use the API function **FileGatheringRename**().

CAUTION

The functions GatheringStopAndSave() and GatheringExternalStopAndSave() overwrite any older files with the same name. After a data gathering, it is required to rename (use the API function FileGatheringRename() to rename the gathering file) or better, to download to a PC using the XPS webpage Files -> Gathering files.

A gathering file can have a maximum of 1,000,000 data entries and a maximum of 25 different data types. The first line of the data file contains the sample period in seconds (minimum period = CorrectorISRPeriod), the second line contains the names of the data type(s) and the other lines contain the acquired data. A sample file is shown below.

Gathering.dat

SamplePeriod	0	0
GatheringTypeA	GatheringTypeB	GatheringTypeC
ValueA1	ValueB1	ValueC1
ValueA2	ValueB2	ValueC2
…	…	…
ValueAN	ValueBN	ValueCN

NOTE

Refer to Programmer's manual to get a list of gathering data types and a list of external gathering data types.

8.1 Time-Based (Internal) Data Gathering

The data for time-based gathering are latched by an internal interrupt related to the servo cycle of the XPS. The function GatheringConfigurationSet() defines the type of data that will be stored in the data file. The following is a list of all the data type(s) that can be collected:

PositionerName.CurrentPosition

PositionerName.SetpointPosition

PositionerName.FollowingError

PositionerName.CurrentVelocity

PositionerName.SetpointVelocity

PositionerName.CurrentAcceleration

PositionerName.SetpointAcceleration

PositionerName.CorrectorOutput

GPIO (ADC, DAC, DI, DO) See the Programmer's Manual for a list of all the GPIO Names of the Analog and Digital I/O.

The Setpoint values refer to the theoretical values from the profiler whereas the current values refer to the actual or real values of position, velocity and acceleration.

It is possible to start the gathering either by a function call or at an event. The following sequence of functions is used for a time-based data gathering started by a function call:

GatheringConfigurationSet()

GatheringRun()

The following sequence of functions is used to start a time-based data gathering at an event:

GatheringConfigurationSet()

EventExtendedConfigurationTriggerSet()

EventExtendedConfigurationActionSet()

EventExtendedStart()

A function triggers the action, for instance, a GroupMoveRelative().

When all data is gathered, use the function **GatheringStopAndSave**() to save the data from the buffer to the flash disk of the XPS controller. To rename the gathering file use the API function **FileGatheringRename**().

Other functions associated with internal Gathering are:

GatheringConfigurationGet()

GatheringCurrentNumberGet()

GatheringDataGet()

GatheringDataMultipleLinesGet()

GatheringStop()

GatheringRunAppend()

See the Programmer's Manual for details about these functions.

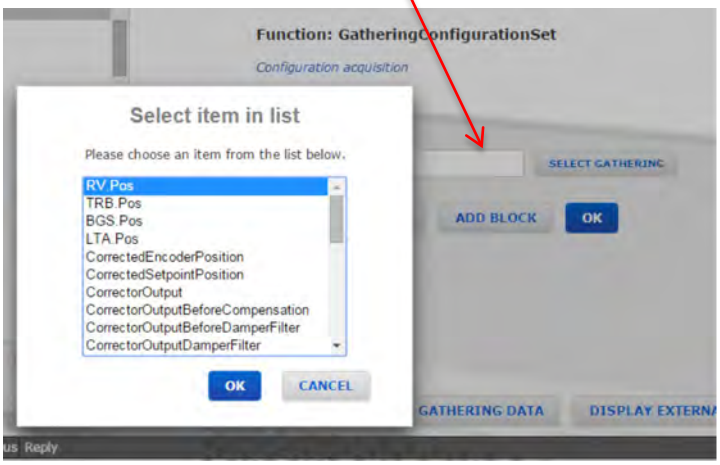
NOTE

When using the function GatheringConfigurationSet() from the terminal screen of the XPS utility, the syntax for one parameter is not directly accessible. For instance, for the parameter XY.X.SetpointPosition, first select XY.X from the choice list. Then, click on the choice field again and select SetpointPosition. See also screen shots on the next page.

For specifying more than one data type, use the ADD button. Select the next parameter as described below.

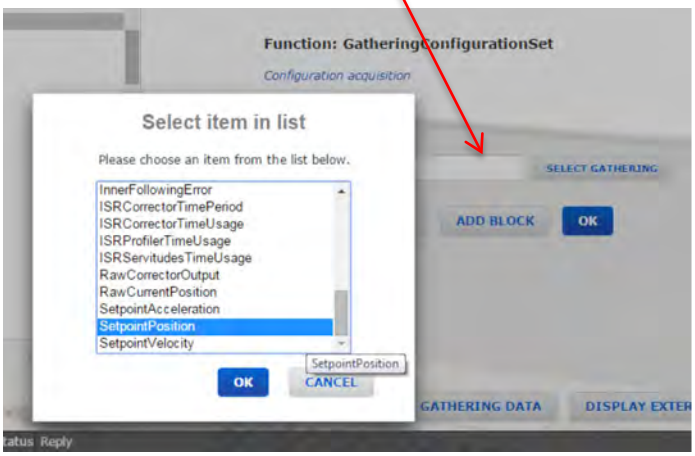
Step 1:

Click **"SELECT GATHERING"** then select the positioner name and click **"OK"**.



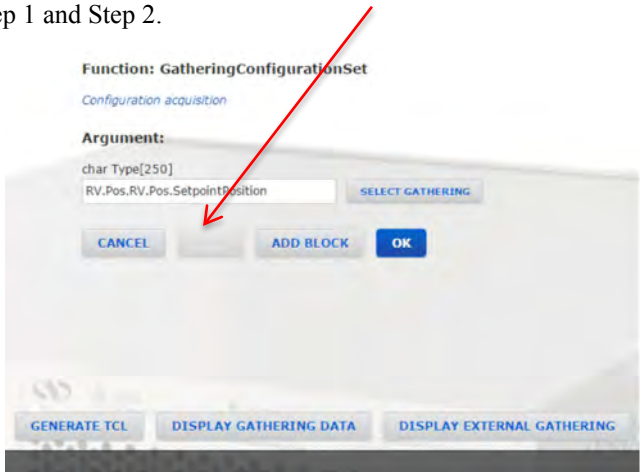
Step 2:

Click **"SELECT GATHERING"** again then select the parameter name and click **"OK"**.



Step 3:

To add another parameter, click **"ADD BLOCK".**
Repeat Step 1 and Step 2.



Example 1

Using the terminal screen of the XPS utility, this example shows the sequence of functions to accomplish a time-based data gathering triggered at an event.

GroupInitialize(XY)

GroupHomeSearch(XY)

GatheringConfigurationSet(XY.X.SetpointPosition, XY.X.CurrentVelocity, XY.X.SetpointAcceleration)

The 3 data XY.X.SetpointPosition, XY.X.CurrentVelocity and XY.X.SetpointAcceleration will be gathered.

EventExtendedConfigurationTriggerSet (XY.X.SGamma.MotionStart,0,0,0,0)

EventExtendedConfigurationActionSet(GatheringRun,4000,8,0,0)

EventExtendedStart()

GroupMoveRelative(XY.X, 50)

GatheringStopAndSave()

In this example, gathering is started when the positioner XY.X starts its next motion using the Sgamma profiler, in this case with GroupMoveRelative() or possibly with GroupMoveAbsolute(). The types of data being collected are the Setpoint Position, Current Velocity and Setpoint Acceleration for the positioner XY.X. A total of 4000 data sets is collected, one data point every 8th servo cycles.

Example 2

Using the terminal screen of the XPS utility, this example shows the sequence of functions to accomplish a time-based data gathering started by a function call.

GroupInitialize(X)

GroupHomeSearch(X)

GatheringConfigurationSet(X.X.SetpointPosition, X.X.FollowingError)

GatheringRun (5000,8)

GroupMoveRelative (X, 10)

GatheringStop()

GatheringStopAndSave()

In this example, gathering is started by a function call. The SetpointPosition and FollowingError of the positioner XY.X are gathered (every 8th servo cycle). Data gathering is stopped after the relative move is completed.

Gathering will stop automatically once the number of points specified has been collected. However, data will not be saved automatically to a file. The function **GatheringStopAndSave**() must be used to save the data to a file.

It is also possible to halt data gathering at an event. To do so, define another event trigger and assign the action GatheringStop to that event. Use another event trigger and assign the action GatheringRunAppend to continue with gathering. For details, see chapter 7.0: “Event Triggers“.

NOTE

The function GatheringRun() always starts a new internal data gathering and deletes any previous internal gathering data hold in the buffer. If you want to append data to the file, use the function GatheringRunAppend() instead.

8.2 Event-Based (Internal) Data Gathering

The event-based gathering provides a method to gather data at an event. For instance, gathering data at a certain value of a digital or analog input, during a constant velocity state of a motion or on a trajectory pulse.

The event-based data gathering uses the same file as the time-based and the function based data gathering (see sections 8.2 and 8.4). However, unlike the time-based gathering, the event-based gathering appends data to the existing file in memory. This allows gathering of data during several periods or even with different methods in one common file, see examples. To start data gathering in a new file, use the function GatheringReset(), which deletes the current gathering file from memory.

The data type(s) that can be collected with event-based gathering are the same as data for time-based and function-based gathering:

PositionerName.CurrentPosition

PositionerName.SetpointPosition

PositionerName.FollowingError

PositionerName.CurrentVelocity

PositionerName.SetpointVelocity

PositonerName.CurrentAcceleration

PositionerName.SetpointAcceleration

PositionerName.CorrectorOutput

GPIO (ADC, DAC, DI, DO) See Programmer’s manual for a list of all the GPIO Names for the Analog and Digital I/O.

The Setpoint values refer to the theoretical values from the profiler where as the current values refer to the actual or real values of position, velocity and acceleration.

The following sequence of functions is used in event-based data gathering:

GatheringReset()

GatheringConfigurationSet()

EventExtendedConfigurationTriggerSet()

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

EventExtendedStart()

…

Use the function GatheringStopAndSave() to store the gathered file from the buffer to the flash disk of the XPS controller.

Other functions associated with the event-based gathering are:

GatheringConfigurationGet()

GatheringCurrentNumberGet()

GatheringDataGet()

FileGatheringRename()

Please refer to the programmer's manual for details.

Example 1

GatheringReset()

Deletes gathering buffer in memory.

GatheringConfigurationSet(XY.X.CurrentPosition, XY.Y.CurrentPosition, GPIO4.ADC1)

The 3 data XY.X.CurrentPosition, XY.Y.CurrentPosition and GPIO4.ADC1 will be gathered.

EventExtendedConfigurationTriggerSet(GPIO4.ADC1.ADCHighLimit, 5,0,0,0)

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

EventExtendedStart()

Data gathering starts when the value of the GPIO4.ADC1 exceeds 5 Volts. One set of data will be gathered at each servo cycle (as the event is checked at each servo cycle). Data gathering automatically stops when the value of the GPIO4.ADC1 falls below 5 V again and the event is automatically removed (see chapter 7.0: "Event Triggers" for details).

Example 2

TimerSet(Timer1, 8)

Sets the timer 1 to 8 servo ticks.

GatheringReset()

Deletes gathering buffer from memory.

GatheringConfigurationSet(XY.X.CurrentPosition, XY.Y.CurrentPosition, GPIO4.ADC1)

The 3 data XY.X.CurrentPosition, XY.Y.CurrentPosition and GPIO4.ADC1 will be gathered.

EventExtendedConfigurationTriggerSet(Timer1,0,0,0,0, GPIO4.ADC1.ADCHighLimit,5,0,0,0)

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

EventExtendedStart()

Different from the previous example, here the event ADCHighLimit is linked to the event Timer1. This has two effects. First, the event becomes permanent as the event timer is permanent. Second, one set of data is gathered only every 1 ms (combination of events must be true). For details on the event definition, please see chapter 7.0: "Event Triggers".

As a result, one set of data is gathered every 1 ms whenever the value of the GPIO4.ADC1 exceeds 5.

Example 3

TimerSet(Timer1, 8)

Sets the timer 1 to 8 servo ticks.

GatheringReset()

Deletes gathering buffer from memory.

GatheringConfigurationSet(XYZ.X.CurrentPosition, XYZ.Y.CurrentPosition, XYZ.Z.CurrentPosition)

EventExtendedConfigurationTriggerSet(Timer1,0,0,0,0, XYZ.Spline.TrajectoryState,0,0,0,0)

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

EventExtendedStart()

In this example, during the execution of the next spline trajectory on the group XYZ, one set of data will be gathered every 1 ms. In contrast to time-based gathering, which allows programming of a similar function, data gathering will automatically stop at the end of the trajectory. Also, it is not needed to define the total number of data sets that will be gathered.

8.3 Function-Based (Internal) Data Gathering

Function-based gathering provides a method to gather one set of data using a function. It uses the same data file as the time-based and the event-based data gathering, see chapter 9.1 for details. At receipt of the function, one set of data is appended to the gathering file in memory.

The data type(s) that can be collected with the event-based gathering are the same as for the time based and the event-based gathering, see section 8.2 and 8.3 for details.

Example

GatheringReset()

Deletes gathering buffer.

GatheringConfigurationSet(XY.X.CurrentPosition, XY.Y.CurrentPosition)

The 2 data XY.X.CurrentPosition and XY.Y.CurrentPosition will be gathered.

GatheringDataAcquire()

Gathers one set of data.

GatheringCurrentNumberGet()

This function will return 1, 500000; 1 set of data acquired, max. 500 000 sets of data can be acquired.

GatheringDataAcquire()

GatheringDataAcquire()

GatheringCurrentNumberGet()

This function will return 3, 500000; 3 sets of data acquired, max. 500,000 sets of data can be acquired.

8.4 Trigger-Based (External) Data Gathering

The trigger-based data gathering allows acquiring position and analog input data at receipt of an external trigger input (Extended GPIO synchronization inputs GPIO5.DI_15 and GPIO5.DI_16 of the XPS, see **Start-Up Manual** for more details).

The position data is latched by dedicated hardware. The jitter between the trigger signal and the acquisition of the position data is less than 50 ns. The analog inputs, however, are only latched by an internal interrupt at the servo rate and the XPS will store the most recent value. Hence, the acquired analog input data might be as old as the CorrectorISRPeriod.

NOTE

There must be a minimum time one Corrector Period between two successive trigger inputs.

The data of the trigger-based (external) data gathering is stored in a file named ExternalGathering.dat, which is different from the file used for the internal data gathering (Gathering.dat). Hence, internal and external data gathering can be used at the same time.

The function GatheringExternalConfigurationSet() defines which type of data will be gathered and stored in the data file. The following data types can be collected:

PositionerName.ExternalLatchPosition

These positions refer to the uncorrected encoder position, meaning no error corrections are taken into account. For devices with RS422 differential encoders, the resolution of the position information is equal to the encoder resolution.

For devices with sine/cosine 1 Vpp analog encoder interface, the resolution is equal to the encoder scale pitch divided by the value of the positioner hard interpolator, see function PositionerHardInterpolatorFactorGet(). Its value is set to 20 by default; the maximum allowed value is 200. Please refer to the Programmer's Manual for details.

The external latch positions require that the device has an encoder. No position data can be latched with this method for devices that have no encoder.

GPIO4.ADC1 to GPIO4.ADC8 (referring to the 8 analog input channels on GPIO4)

The following sequence of functions is used for a trigger-based data gathering:

GatheringExternalConfigurationSet()

EventExtendedConfigurationTriggerSet()

EventExtendedConfigurationActionSet()

EventExtendedStart()

Other functions associated with trigger-based gathering are:

GatheringConfigurationGet()

GatheringCurrentNumberGet()

GatheringExternalDataGet()

FileGatheringRename()

Please refer to the Programmer's Manual for details.

Example

GatheringExternalConfigurationSet(XY.X.ExternalLatchPosition, GPIO4.ADC1)

EventExtendedConfigurationTriggerSet(Immediate,0,0,0,0)

EventExtendedConfigurationActionSet(ExternalGatheringRun,100,1,0,0)

EventExtendedStart()

In this example, a trigger-based (external) gathering is started immediately (with the function EventExtendedStart()). The types of data being collected are the XY.X encoder position and the value of the GPIO4.ADC1. A total of 100 data sets are collected; one set of data at each trigger input. Gathering will stop automatically after the 100th data acquisition. Use the function **GatheringExternalStopAndSave**() to save the data to a file. The file format is the same as for internal data gathering. To rename the gathering file use the API function **FileGatheringRename**().

9.0 Output Triggers

External data acquisition tools, lasers, and other devices can be synchronized to the motion. For this purpose, the XPS features one dedicated trigger output for Axis 1 and Axis 2, see **Start-Up Manual** for details. The XPS can be configured to either output distance spaced pulses, AquadB encoder signals, or time spaced pulses on this connector.

In the distance spaced configuration, one output pulse is generated when crossing a defined position and a new pulse is generated at every defined distance until a maximum position has been reached. In most cases, this mode provides the most precise synchronization of the motion to an external tool.

In the AquadB configuration, AquadB encoder signals are output on the PCO connector. These signals can be provided either always or only if the positioner is within a defined position window. When used with stages that feature a digital encoder (AquadB) as opposed to a SinCos encoder (AnalogInterpolated), the AquadB configuration essentially provides an image of the encoder signals on the PCO connector.

In the time flasher configuration, an output pulse is generated when crossing a defined position and a new pulse is generated at a defined time interval until a maximum position has been reached. In some cases, this mode can provide an even more precise synchronization of the motion to an external tool, in particular if the variation of the speed multiplied with the time interval is smaller than the error of the encoder signals during the same period.

Dedicated hardware is used to check the position crossing and the time interval to attain less than 50 ns latency between the position crossing and the trigger output.

For the distance spaced pulses configuration, time flasher configuration or AquadB signals on PCO connector configuration, it is recommended to calibrate the position compare before all PCO pulses generation. It is also recommended to set the position compare hardware to the scanning range you intend to use to get the best performances (*refer to section 9.4: "*Distance, Time Spaced Pulses or AquadB Position Compare*" for details*).

In addition and independent from the above, the XPS controller can output distance spaced pulses on Line-arc trajectories and time spaced pulses on PT and PVT trajectories. In these cases, the distances/time intervals are checked on the servo cycle.

9.1 Triggers on Line-Arc Trajectories

This capability outputs pulses at constant trajectory length intervals on Line-Arc-Trajectories. The pulses are generated between a start length and an end length. All lengths are calculated in an orthogonal XY plane. The StartLength, EndLength, and PathLengthInterval refer to the Setpoint positions.

The trajectory length is calculated at the servo rate. The trajectory length is = **CorrectorISRPeriod** * trajectory velocity. If the programmed PathLengthInterval is not a multiple of this resolution, the pulses can be off from the ideal positions by a maximum ± half of this resolution.

Two signals are provided for each XPS-D GPIO option:

GPIO Basic:

GPIO3.DO6, Window:	A constant 5 V signal is sent between the StartLength and the EndLength.
GPIO3.DO7, Pulse:	A 1µs pulse with 5 V peak voltage is sent every PathLengthInterval.

GPIO Extended:

GPIO5.DO14, Window: A constant 5 V signal is sent between the StartLength and the EndLength.

GPIO5.DO15, Pulse: A 1µs pulse with 5 V peak voltage is sent every PathLengthInterval.

For details about the XPS I/O connectors, see **Start-Up Manual**.

To define the StartLength, EndLength, and PathLengthInterval, use the function **XYLineArcPulseOutputSet**().

Example

XYLineArcPulseOutputSet(XY, 10, 30, 0.01)

One pulse will be generated every 10 µm on the next Line-Arc Trajectory between 10 mm and 30 mm.

XYLineArcVerification(XY, Traj.trj)

Loads and verifies the trajectory Traj.trj

XYLineArcExecution(XY, Traj.trj, 10, 100, 1)

Executes the trajectory at a trajectory speed of 10 mm/s and with a trajectory acceleration of 100 mm/s one time.

Please note, that the pulse output settings are automatically removed when the trajectory is over. Hence, with the execution of every new trajectory, it is also required to define the pulse output settings again.

It is also possible to use the trajectory pulses and the pulse window state as events in the event triggers (see chapter 7.0: “Event Triggers“ for details). This allows the gathering of data on a trajectory at constant length intervals.

Example

XYLineArcPulseOutputSet(XY, 10, 30, 0.01)

One pulse every 10 µm will be generated on the Line-Arc Trajectory between 10 mm and 30 mm.

XYLineArcVerification(XY, Traj.trj)

Loads and verifies the trajectory Traj.trj

GatheringConfigurationSet(XY.X.CurrentPosition, XY.Y.CurrentPosition, GPIO4.ADC1)

Configures data gathering to capture the current positions of the XY.X and the XY.Y and the analog input GPIO4.ADC1

EventExtendedConfigurationTriggerSet(Always, 0,0,0,0,XY.LineArc.TrajectoryPulse,0,0,0,0)

Triggers an action for every trajectory pulse. The link of the event TrajectorPulse with the event Always is important to make the event permanent. Otherwise, the event will be removed after the first pulse.

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

Defines the action; gathers one set of data each trajectory pulse.

EventExtendedStart()

Starts the event trigger.

XYLineArcExecution(XY, Traj.trj, 10, 100, 1)

Executes the trajectory at a trajectory speed of 10 mm/s and a trajectory acceleration of 100 mm/s one time.

GatheringStopAndSave()

Saves the gathering data from memory into a file gathering.dat in the ..Admin/Public/Gathering folder of the XPS.

In this example, one set of data will be gathered on the trajectory between length 10 mm and 30 mm at constant trajectory length intervals of 10 µm.

9.2 Triggers on PVT Trajectories

This capability outputs pulses at constant time intervals on a PVT trajectory. The pulses are generated between a first and a last trajectory element (see section 4.3: PVT Trajectories for details). The minimum possible time interval is one servo cycle.

Two signals are provided for each XPS-D GPIO option:

GPIO Basic:

GPIO3.DO6, Window:	A constant 5 V signal is sent between the StartLength and the EndLength.
GPIO3.DO7, Pulse:	A 1µs pulse with 5 V peak voltage is sent every PathLengthInterval.

GPIO Extended:

GPIO5.DO14, Window:	A constant 5 V signal is sent between the StartLength and the EndLength.
GPIO5.DO15, Pulse:	A 1µs pulse with 5 V peak voltage is sent every PathLengthInterval.

For details about the XPS I/O connectors, see **Start-Up Manual**.

To define the first element, the last element and the time interval, use the function **MultipleAxesPVTPulseOutputSet**().

Example 1

MultipleAxesPVTPulseOutputSet(Group1, 3, 5, 0.01)

One pulse will be generated every 10 ms between the start of the 3rd element and the end of the 5th element.

MultipleAxesPVTVerification(Group1, Traj.trj)

Loads and verifies the trajectory Traj.trj

MultipleAxesPVTExecution(Group1, Traj.trj, 1)

Executes the trajectory Traj.trj one time.

Note that the pulse output settings are automatically removed when the trajectory is over. Hence, with the execution of every new trajectory, the pulse output settings must be defined again.

It is also possible to use the trajectory pulses and the pulse window state as events in the event triggers (see chapter 7.0: “Event Triggers” for details). This allows the gathering of data on a trajectory.

Example 2

MultipleAxesPVTPulseOutputSet(Group1, 3, 5, 0.01)

One pulse will be generated every 10 ms between the start of the 3rd element and the end of the 5th element.

MultipleAxesPVTVerification(Group1, Traj.trj)

Loads and verifies the trajectory Traj.trj

GatheringConfigurationSet(Group1.P.CurrentPosition, GPIO4.ADC1)

Configures data gathering to capture the current position of the Group1.P positioner and the analog input GPIO4.ADC1

EventExtendedConfigurationTriggerSet(Always, 0,0,0,0,Group1.PVT.TrajectoryPulse,0,0,0,0)

Triggers an action for every trajectory pulse. The link of the event TrajectorPulse with the event Always is important to make the event permanent. Otherwise, the event will be removed after the first pulse.

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

Defines the action; gathers one set of data each trajectory pulse.

EventExtendedStart()

Starts the event trigger

MultipleAxesPVTExecution(XY, Traj.trj, 1)

Executes the trajectory Traj.trj one time.

GatheringStopAndSave()

Saves the gathering data from memory in a file gathering.dat in the ..Admin/Public/Gathering folder of the XPS.

In this example, one set of data will be gathered every 10 ms on the trajectory between the start of the 3rd and the end of the 5th element.

9.3 Triggers on PT Trajectories

This capability outputs pulses at constant time intervals on a PT trajectory. The pulses are generated between a first and a last trajectory element (see section 8.4: PT Trajectories for details). The minimum possible time interval is one servo cycle.

Provided signals are:

GPIO Extended:

GPIO3.DO6, Window:	A constant 5 V signal is sent between the StartLength and the EndLength.
GPIO3.DO7, Pulse:	A 1µs pulse with 5 V peak voltage is sent every PathLengthInterval.

GPIO Basic:

GPIO5.DO14, Window:	A constant 5 V signal is sent between the StartLength and the EndLength.
GPIO5.DO15, Pulse:	A 1µs pulse with 5 V peak voltage is sent every PathLengthInterval.

For details about the XPS I/O connectors, see Appendix B: General I/O Description.

To define the first element, the last element and the time interval, use the function **MultipleAxesPTPulseOutputSet**().

Example 1

MultipleAxesPTPulseOutputSet (MultipleGroup, 3, 5, 0.01)

One pulse will be generated every 10 ms between the start of the 3rd element and the end of the 5th element.

MultipleAxesPTVerification(MultipleGroup, Traj.trj)

Loads and verifies the trajectory Traj.trj

MultipleAxesPTExecution(MultipleGroup, Traj.trj, 1)

Executes the trajectory Traj.trj one time.

Note that the pulse output settings are automatically removed when the trajectory is over. Hence, with the execution of every new trajectory, the pulse output settings must be defined again.

It is also possible to use the trajectory pulses and the pulse window state as events in the event triggers (see chapter 11.0: "Event Triggers" for details). This allows the gathering of data on a trajectory.

Example 2

MultipleAxesPTPulseOutputSet(MultipleGroup, 3, 5, 0.01)

One pulse will be generated every 10 ms between the start of the 3rd element and the end of the 5th element.

MultipleAxesPTVerification(Group1, Traj.trj)

Loads and verifies the trajectory Traj.trj

GatheringConfigurationSet(MultipleGroup.Pos1.CurrentPosition, GPIO4.ADC1)

Configures data gathering to capture the current position of the MultipleGroup.Pos1 positioner and the analog input GPIO4.ADC1

EventExtendedConfigurationTriggerSet(Always, 0,0,0,0, MultipleGroup.PT.TrajectoryPulse,0,0,0,0)

Triggers an action for every trajectory pulse. The link of the event TrajectorPulse with the event "Always" is important to make the event permanent. Otherwise, the event will be removed after the first pulse.

EventExtendedConfigurationActionSet(GatheringOneData,0,0,0,0)

Defines the action; gathers one set of data each trajectory pulse.

EventExtendedStart()

Starts the event trigger

MultipleAxesPTExecution(MultipleGroup, Traj.trj, 1)

Executes the trajectory Traj.trj one time.

GatheringStopAndSave()

Saves the gathering data from memory in a file "gathering.dat" in the "Admin/Public/Gathering" folder of the XPS.

In this example, one set of data will be gathered every 10 ms on the trajectory between the start of the 3rd and the end of the 5th element.

9.4 Distance, Time Spaced Pulses or AquadB Position Compare

9.4.1 Even Distance Spaced Pulses Position Compare

In the even distance spaced pulse configuration, one first output pulse is generated when the positioner enters the defined position window. This is independent of the positioner entering the window from the minimum position or from the maximum position. From this first pulse position, a new pulse is generated at every position step until the stage exits the window.

NOTE

To make sure that the trigger pulses are always at the same positions independent of the positioner entering the window from the minimum or from the maximum window position, the difference between the minimum and the maximum window position should be an integer multiple of the position step.

The duration of the trigger pulse is 2 µs by default and can be modified using the function **PositionerPositionComparePulseParametersSet (PositionerName, PCOPulseWidth)**. Possible values for PCOPulseWidth are: 35 ns to 327.68 µs (5 ns resolution). Successive trigger pulses should have a minimum time lag of 625 ns (200 ns for less than 4096 pulses).

The following functions are used to configure the distance spaced pulses:

PositionerPositionCompareSet

PositionerPositionCompareGet

PositionerPositionCompareEnable

PositionerPositionCompareDisable

The function PositonerPositonCompareSet() defines the position window and the distance for the trigger pulses. It has four input parameters:

Positioner Name

Minimum Position

Maximum Position

Position Step

To enable the distance spaced pulses, the function PositionerPositionCompareEnable() must be sent.

Example

GroupInitialize(MyStage)

GroupHomeSearch(MyStage)

PositionerPositionCompareSet(MyStage.X,5, 25, 0.002)

PositonerPositionCompareEnable(MyStage.X)

PositionerPositionCompareGet(MyStage, &MinimumPosition, &MaximumPosition, &PositionStep, &EnableState)

This function returns the parameters previously defined, the minimum position 5, the maximum position 25, the position step 0.002 and the enabled state (1=enabled, 0 =disabled).

GroupMoveAbsolute(MyStage,30)

PositionerPositionCompareDisable(MyStage.X)

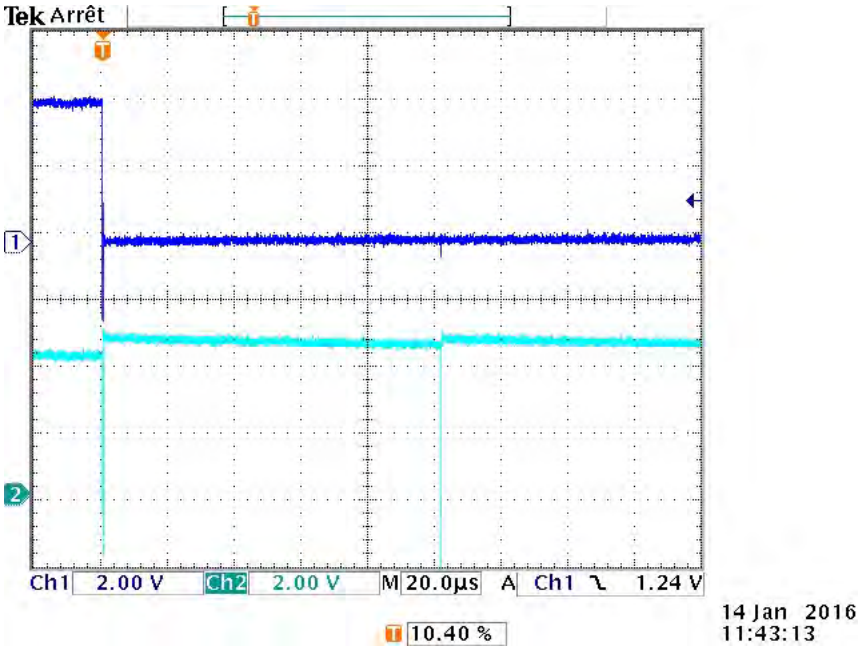
The group has to be in a READY state for the position compare to be enabled. Also, the PositionerPositionCompareSet() function must be completed before the PositionerPositionCompareEnable() function. In this example, one trigger pulse is generated every 0.002 mm between the minimum position of 5 mm and the maximum position of 25 mm. The first trigger pulse will be at 5 mm and the last trigger pulse will be at 25 mm.

The output pulses are accessible from the PCO connector at the back of the XPS controller, see **Start-Up Manual** for details.

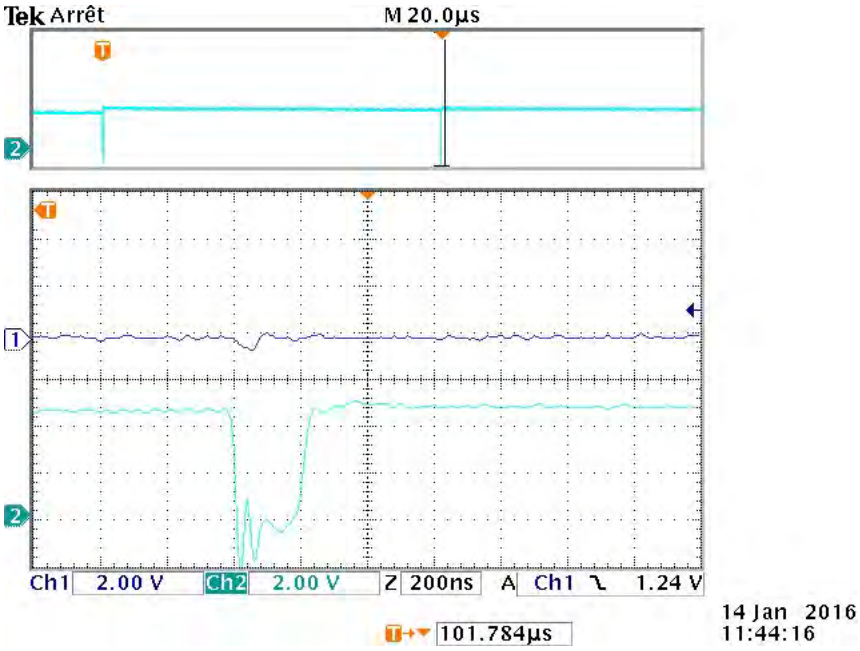
This table summarizes the results of the example above:

Position of the stage	Pulse enable 1 state	Pulse 1 activation	Explanation
0	0	No	Position compare not enabled
5	1	Yes	Position compare enabled, first pulse
5…25	1	Yes	One pulse every 0.002 mm
25	1	Yes	Last pulse
25.002	0	No	Position compare disabled
30	0	No	Position compare disabled

The figure below shows actual screen shots from an oscilloscope for the example above. The enable window is displayed in ch1 and the pulses in ch2:



At position 5 mm, the position compare output functionality becomes active and the first pulse is generated. Then, pulses are generated every 2 µm which equals a time span of 100 µs at a speed of 20 mm/s (2 µm/20 mm/s = 100 µs).



This second picture shows a zoom of the second pulse. The duration of the pulse should be 200 ns.

NOTE

The parameters PositionStep, MinimumPosition, and MaximumPosition (specified with the function PositionerPositionCompareSet) are rounded to the nearest detectable trigger position. When using the Position Compare function with AquadB encoders, the trigger resolution is equal to the EncoderResolution of the positioner specified in the stages.ini. When using the Position Compare function with AnalogInterpolated encoders, the trigger resolution is equal to the EncoderScalePitch defined in the stages.ini divided by 256 (basic PCO) or 65536 (extended PCO).

AnalogInterpolated encoder

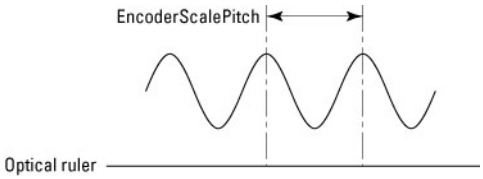


Figure 28: Analog interpolated encoder.

$$\textbf{Trigger resolution} = \frac{\text{EncoderScalePitch}}{\text{InterpolationFactor}}$$

InterpolationFactor = 256 with basic PCO or 65536 with extended PCO.

Trigger pulses

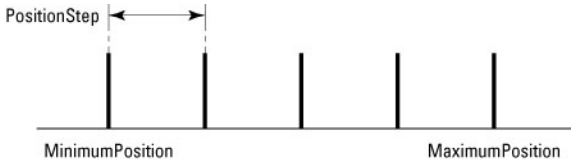


Figure 29: Trigger pulses.

MinimumPosition, MaximumPosition, and PositionStep should be multiples of the Trigger resolution. If not, rounding to the nearest multiple value is made.

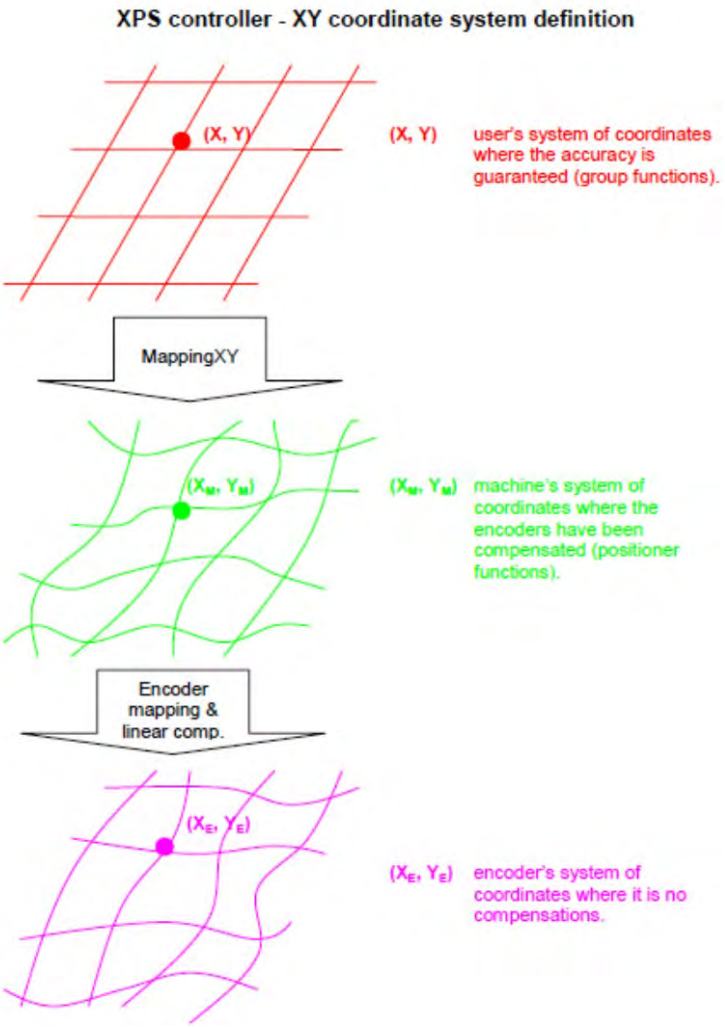
9.4.2 Compensated Position Compare

This feature is used to output a pulse each time the stage moves over user predefined positions compensated through error mapping. Available only for a XPS with the Extended PCO feature.

9.4.3 XPS System of Coordinates

To explore the details of the XPS coordinate system, use the example of the XY group but the same is true for the other groups.

The firing positions are defined in the called user's system of coordinates (X, Y). The controller will convert the (X, Y) coordinates to raw encoder positions (X_E, Y_E) to take into account the group mapping, the encoder mapping and the encoder linear compensation to accurately fire the pulses at the requested positions.



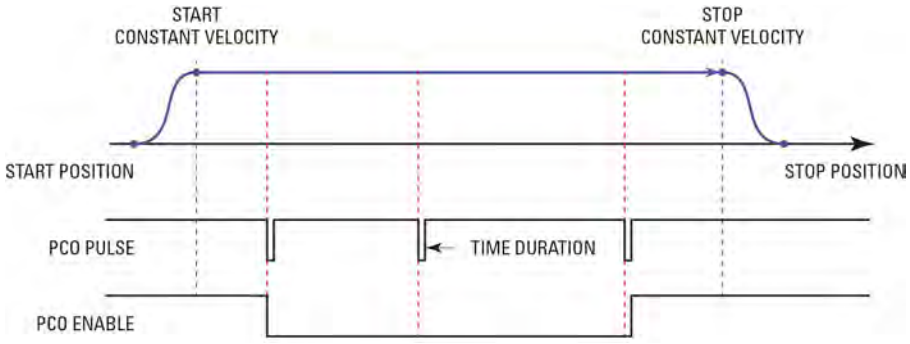
To know the positions in the different systems of coordinates, the following functions are provided:

- *GroupPositionCorrectedProfilerGet()* function has as input a (X, Y) position in the user's system of coordinates and will output the (X_M, Y_M) position in the machine's system of coordinates by applying the XY mapping compensation.
- *XYGroupPositionPCORawEncoderGet()* function has as input a (X, Y) position in the user's coordinate system and will output the (X_E, Y_E) position in the encoder's system of coordinates without any compensation.

9.4.4 Compensated Position compare signals definition

If the compensated PCO pulses generation is activated, the PCO pulses will be generated at each predefined position with a pulse time duration that can be set with the *PositionerCompensatedFastPCOPulseParametersSet()* function *(cf. XPS Programmer's Manual for details).*

The PCO enable will be generated from the beginning of the first pulse to the end of the last pulse.



The pulse rate is limited by 2 things:

- The minimum time between 2 successive pulse : 200 ns
- The transfer rate of the target positions from CPU memory to CIE internal memory (4096 positions FIFO) : 1.6 MHz

If there are less than 4096 pulses to generate, the only limitation is the time between 2 successive points and the maximum pulses frequency is 5 MHz.

*MinimumTriggerPulseDistance > ScanningVelocity * 200 ns*

For more than 4096 pulses the limitation is the transfer frequency and the maximum frequency is 1.6 MHz.

*MinimumTriggerPulseDistance > ScanningVelocity * 625 ns*

The margin to take in account will depend on many parameters such as the speed stability.

If there are two PCO running on the same CIE board, the maximum transfer frequency of 1.6 MHz is divided by 2, but the minimum time between 2 successives pulses when there are less than 4096 points is still 200 ns.

If the transfer rate limitation is reached, the PCO error flag will be set.

If the time between two successive pulse is too short, the position will not be detected and the pulse generation will stop.

9.4.5 Compensated Position compare scanning process description

Scan preparation

- Initialize and home the scanning group:

  GroupInitialize(Group) *; Initialize scanning group*

  GroupHomeSearch(Group) *; Search home for the scanning*

Scan execution

- If needed, set PCO pulse duration and polarity/type (pulse or toggle):

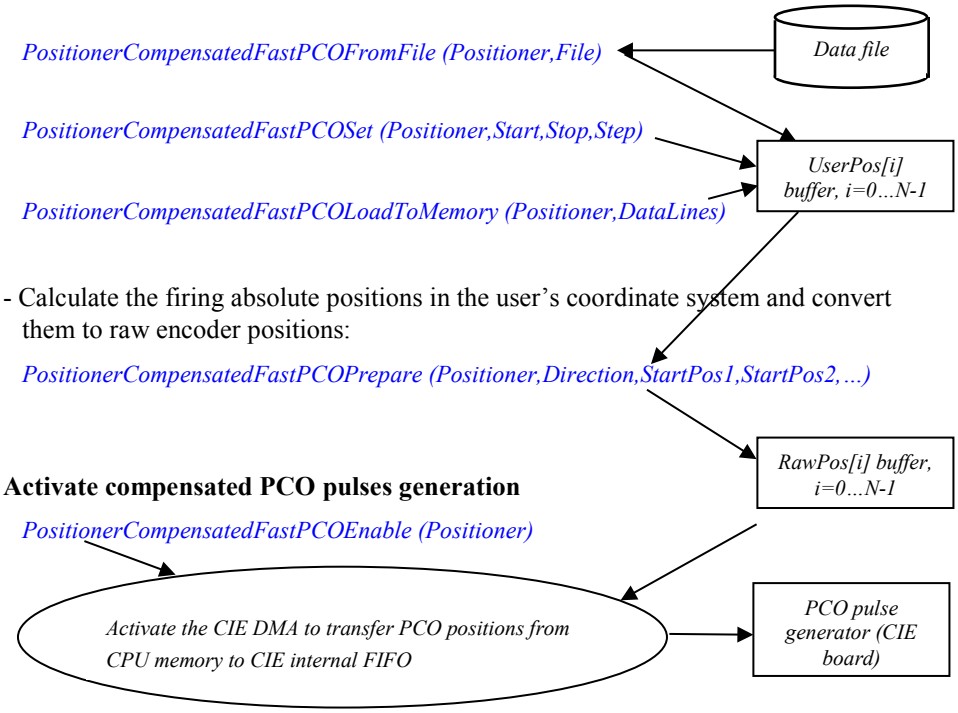
  PositionerCompensatedFastPCOPulseParametersSet(Positioner,PulseDuration,PulsePolarity,PulseToggle)

- Move the scanning group to the scan start position (*outside of the scanning zone*):

  GroupMoveAbsolute (Group, Position1, Position2, ...)

- Set the firing positions by reading data from file or loading to controller's memory or with a "set" function.

  Note that the firing positions defined with the following functions are only the offsets relative to the scanning positioner start position, that will be specified with the *PositionerCompensatedFastPCOPrepare()* function.



- Set motion parameters for scan:

  PositionerSGammaParametersSet(Positioner, ScanVelocity, ScanAcceleration, MinimumJerkTime,MaximumJerkTime)

- Move the scanning positioner across the scanning zone, during this move the firing pulses will be generated:

  GroupMoveRelative (Positioner, ScanDistance)

9.4.6 Compensated Position Compare Related Functions

Here is the list of the associated functions with a brief description. For detailed information, refer to the *XPS Programmer's Manual*.

Firing positions definition

There are three ways to enter the firing positions: reading from file, writing directly to the controller's memory or calculating with a "set" function.

- Firing positions definition from a data file

Function *PositionerCompensatedFastPCOFromFile(Positioner, FileName)* reads firing positions from a data file to the controller's memory.

- Firing positions definition from a "load to memory" function

Function *PositionerCompensatedFastPCOFromFile(Positioner, DataLines)* appends firing positions to the controller's memory from *DataLines* parameter.

To reset the controller's memory, the *PositionerCompensatedFastPCOMemoryReset()* function is provided.

- Firing positions definition from a "set" function

Function *PositionerCompensatedFastPCOSet (Positioner,Start,Stop,Step)* calculates a set of evenly spaced firing positions to the controller's memory.

Firing positions preparation

Function *PositionerCompensatedFastPCOPrepare (Positioner, ScanDirection, StartPosition1, StartPosition2, ...)* calculates the firing at absolute positions, *in user's coordinate system* and converts them to firing absolute raw PCO positions, *in encoder's coordinate system*.

When mappings are enabled, the correction between the user's coordinate system position and raw encoder position will be different at each different location. For this reason, the prepare function must know the location (*positions of all positioners in the scanning group*) where the scan will be done.

Associated functions

- Pulses generation enable

Function *PositionerCompensatedFastPCOEnable (Positioner)* activates the compensated PCO pulses generation *(status becomes running (value 1))*. The pulses will be generated when the scanning positioner will move across the predefined positions. When the last pulse is generated, the compensated PCO mode will become inactive (*status becomes inactive (value 0))*. To get the status of the compensated PCO pulses generation, use the *PositionerCompensatedFastPCOCurrentStatusGet()* function.

Note that only the scanning positioner positions are used to fire pulses: if you prepare a set of positions at a given location but you enable the firing pulses generation and start the move from a different location, the pulses could be generated but their accuracy will be impacted by the mapping difference between the two locations.

- Pulses generation abort

Function *PositionerCompensatedFastPCOAbort (Positioner)* disables the compensated PCO pulses generation. The pulses generation is stopped immediately; no more pulse will be generated even if the scanning positioner continues to move across the predefined firing positions. To stop the scanning move, use *GroupMoveAbort()* function.

- Pulses data reset

The function *PositionerCompensatedFastPCOMemoryReset (Positioner)* resets the compensated PCO data memory. This function is useful to remove the data that was previously entered with the *PositionerCompensatedFastPCOLoadToMemory()* function.

- Pulses generation status get

The function *PositionerCompensatedFastPCOCurrentStatusGet (Positioner, Status)* gets the current status of compensated PCO pulses generation.

9.4.7 Time Spaced Pulses (Time Flasher)

In the time spaced configuration, a first pulse is generated when the motion axis enters the time pulse window. From this first pulse, a new pulse is generated at every time interval until the positioner exits the time pulse window.

Hardware attains less than 5 ns jitter for the trigger pulses. The duration of the pulse is 2 µs by default and can be modified using the function **PositionerPositionComparePulseParametersSet**(). Possible values for the PCOPulseWidth are: 35 ns to 327.68 µs (5 ns resolution). Successive trigger pulses should have a minimum time lag of 50 ns.

The following functions are used to generate time spaced pulses:

PositionerTimeFlasherSet

PositionerTimeFlasherGet

PositionerTimeFlasherEnable

PositionerTimeFlasherDisable

The function PositonerTimeFlasherSet() defines the position window and the time intervals for the trigger signals. It has four input parameters:

Position Name

Minimum Position

Maximum Position

Time Interval

The time interval must be greater than or equal to 0.00000005 seconds (50 ns) and less than or equal to 21.47483648 seconds. Furthermore, the time interval must be a multiple of 5 ns.

To enable the time spaced pulses, the function PositionerTimeFlasherEnable() must be sent.

Example 1

GroupInitialize(MyStage)

GroupHomeSearch(MyStage)

PositionerTimeFlasherSet(MyStage.X,5, 25, 0.00001)

PositonerTimeFlasherEnable(MyStage.X)

GroupMoveAbsolute(MyStage,30)

PositionerTimeFlasherDisable(MyStage.X)

The group has to be in a READY state for the time flasher to be enabled. Also, the PositionerTimeFlasherSet() function must be completed before the PositionerTimeFlasherEnable() function. In this example, one trigger pulse is generated every 0.00001 seconds or at a rate of 100 kHz between the minimum position of 5 mm and the maximum position of 25 mm. The first trigger pulse will be at 5 mm and the last trigger pulse will be at 25 mm or before.

The output pulses are accessible from the PCO connector at the back of the XPS controller, See **Start-Up Manual** for details.

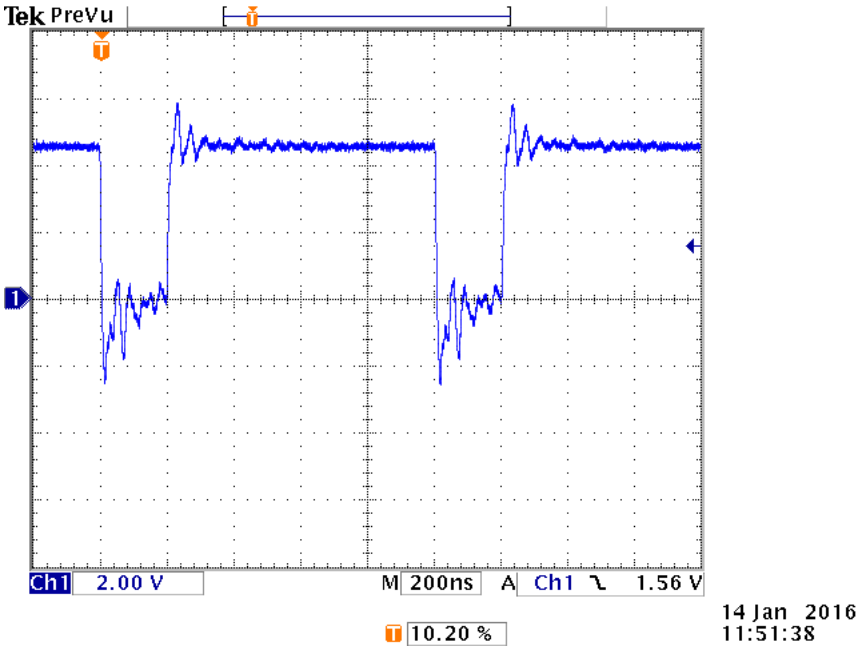


Figure 30: Temporal resolution of time spaced pulses in oscilloscope view.

Example 2

The time flasher function is of particular use with high precision (direct drive) stages. At high speeds, these stages typically provide very good speed stability. In other words, the position change over a short time interval is highly consistent and repeatable. Hence, time spaced pulses can be used for synchronization with similar, in some cases even higher precision as distance spaced pulses. The time spaced pulse configuration, however, provides some further flexibility with regards to the nominal distance between successive triggers.
Consider an XM stage for instance. XM stages feature an analog encoder with 4 µm signal period. The max. resolution of the distance spaced pulses is 15.625 nm (256x interpolation). If the goal is to get pulses at a nominal distance of 272.5 nm at a speed of 200 mm/s speed, this is not possible using the distance spaced pulse configuration. Either 265.625 nm or 281.25 nm are possible, but not 272.5 nm. With some minor adjustments to the target speed, however, this is possible using the time spaced pulse configuration:

- The target speed is 200 mm/s, the desired distance between successive pulses is 272.5 nm. So the nominal time interval between successive pulses is:
  272.5 nm/200 mm/s = 1.3625 µs
- Round this nominal value to the next possible time interval, means to the next integer multiple of 5 ns : 1.365 µs
  Use this rounded time interval to calculate a corrected velocity:
  272.5 nm/1.365 µs = 199.6337 mm/s

  PositionerSGammaVelocityAndAccelerationSet(MyStage.X,199.6337,2500)
  PositionerTimeFlasherSet(MyStage.X, -30, 30, 0.000001365)
  PositionerTimeFlasherEnable(MyStage.X)
  GroupMoveAbsolute(MyStage.X)
  PositionerTimeFlasherDisable(MyStage.X)

In this example, a first pulse is generated when the stage crosses the position -30 mm. Further pulses are generated every 1.350 µs until the stage reaches the maximum position of +30 mm. Since the stage moves at a speed of 199.6337 mm/s, the nominal distance between successive pulses is: 199.6337 mm/s * 1.365 µs = 272.5 nm.

9.4.8 AquadB Signals on PCO Connector

In the AquadB signals configuration, AquadB encoder signals are provided on the PCO connector, see **Start-Up Manual** for details and pinning. These signals are either output always (Always configuration), or only when the positioner is within a defined position window (Windowed configuration).

When used with stages that feature a digital encoder (AquadB), the AquadB signals are the same as the encoder signals of the stage. When used with SinCos encoders (AnalogInterpolated), the resolution of the AquadB signal is defined by the signal period of the encoder and the settings of the prescaler by the function PositionerPositionCompareAquadBPrescalerSet().

Example

XM stages feature an analog encoder with a signal period of 4 µm. With the setting PositionerPositionCompareAquadBPrescalerSet(MyStage.X,256) the post-quadrature resolution of the AquadB signals is: 4 µm/256 = 0.015625 µm. In this case one full period of the AquadB signals equals 0.0625 µm.

The following functions are used to configure AquadB signals:

PositionerPositionCompareAquadBWindowedSet

PositionerPositionCompareAquadBWindowedGet

PositionerPositionCompareEnable

PositionerPositionCompareAquadBAlwaysEnable

PositionerPositionCompareDisable

PositionerPositionCompareAquadBPrescalerSet

The function PositonerPositonCompareAquadBAlwaysEnable() has only one input parameter, the positioner name. When sent, AquadB signals are generated always. To disable this mode use the function PositionerPositionCompareDisable().

The function PositonerPositonCompareAquadBWindowedSet() has three input parameters.

Positioner name

Minimum Position

Maximum Position

To enable the AquadB signals, the function PositionerPositionCompareEnable() must be sent.

Example

GroupInitialize(MyStage)

GroupHomeSearch(MyStage)

PositionerPositionCompareAquadBWindowedSet(MyStage.X, 10, 20)

PositonerPositionCompareEnable(MyStage.X)

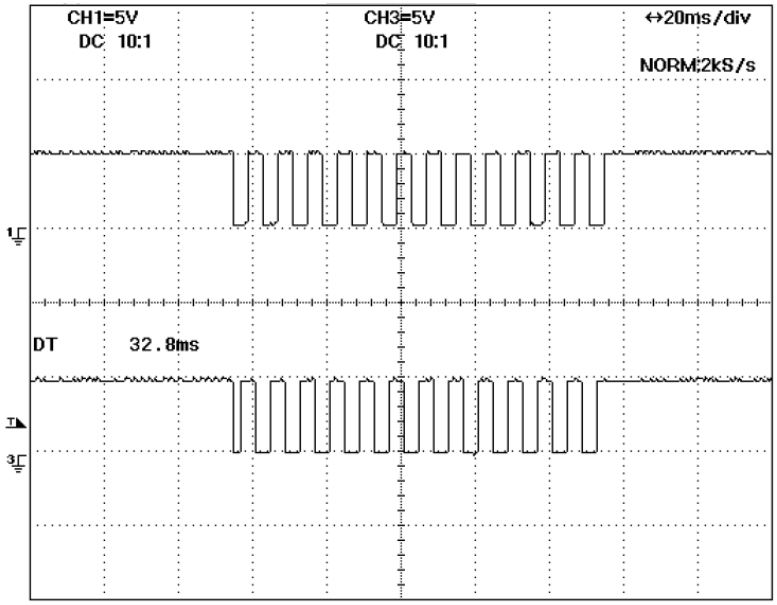
PositionerPositionCompareGet(MyStage, &MinimumPosition, &MaximumPosition, &EnableState)

This function returns the parameters previously defined, the minimum position 10, the maximum position 20 and the enabled state (1=enabled, 0 =disabled).

GroupMoveAbsolute(MyStage,30)

PositionerPositionCompareDisable(MyStage.X)

The figure below shows a screen shots from an oscilloscope for the example above.



The group has to be in a READY state for the position compare to be enabled. Also, the PositionerPositionCompareAquadBWindowedSet() function must be completed before the PositionerPositionCompareEnable() function. In this example, AquadB signals are generated when the positioner is between the minimum position of 10 mm and the maximum position of 20 mm.

NOTE

The AquadB signal configuration is only available with positioners that have an encoder (AquadB or AnalogInterpolated).

The AquadB signals can not be provided at the same time as the distance spaced pulses (PCO) or the time spaced pulses.

The function PositionerPositionCompareEnable() enables always the last configuration sent, either distance spaced pulses defined with the function PositionerPositionCompareSet() or AquadB pulses defined with the function PositionerPositionCompareAquadBWindowedSet().

10.0 Control Loops

10.1 XPS Servo Loops

10.1.1 Servo structure and Basics

The XPS controller can be used to control a wide range of motion devices, which are categorized by the XPS as “positioners”. Within the structure of the XPS' firmware, a “positioner” is defined as an object with an associated profile (trajectory), a PID corrector, a motor interface, a driver, a stage and an encoder.

The general schematic of a positioner servo loop is below.

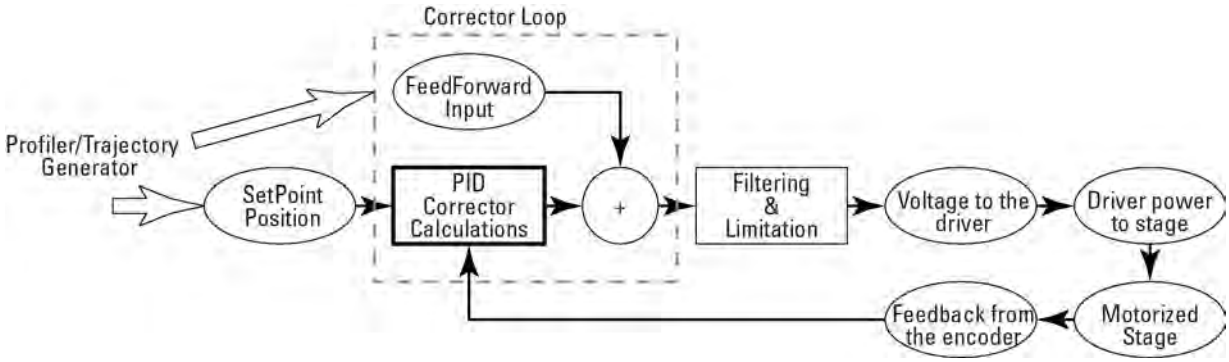


Figure 31: Servo structure and basics.

The calculations done by the “servo loop” result in a voltage output from the controller that is applied to the driver, which can be either any of Newport's Universal drive modules or to an external driver through the XPS pass-through module. Depending on the corrector loop type selected, the level of this output voltage can be the result of two gain factors, the PID corrector and the FeedForward loop. The XPS has imbedded configuration files that provide optimized corrector loop settings for all Newport stages. Non-Newport stages may need to be assigned a specific corrector loop setting during the set-up process. In addition to the two main gain loops the XPS also adds filtering and error compensation parameters to this servo loop to improve system response and reliability.

The profiler (Trajectory Generator) within the controller calculates in real time, the position, velocity, and acceleration/deceleration that the positioner must follow to reach its commanded position (Setpoint Position). This profile is updated at the ProfileGenerator rate. The ProfileGenerator rate is defined in relation to the servo rate given by the expression:

Profile Generator Period = CorrectorISRPeriod * ProfileGenatorISRRatio

The ProfileGenatorISRRatio and the CorrectorISRPeriod values are found in the system.ref file.

Example system.ref file

```
[GENERAL]
OptionalModuleNames =
CorrectorISRPeriod = 125e-6              ; seconds
ProfileGeneratorISRRatio = 4
ServitudesISRRatio = 10
GatheringBufferSize = 1000000            ; data count
DelayBeforeStartup = 0                   ; seconds
DebugTraceCommunicationBufferSize = 0    ; characters, if 0 =>
no trace
```

The PID corrector then compares the SetpointPosition, as defined by the profiler, and the current position, as reported by the positioner's encoder, to determine the current following error. The PID corrector then outputs a value that the controller uses to maintain, increase or decrease the output voltage, which is applied to the driver. This loop is updated at servo rate. The adjustment of the PID parameters allows users to optimize the performance of their positioner or system by increasing or decreasing the responsiveness of the output to increasing or decreasing following errors. Refer to the section 14.3 on PID tuning for more information and tips on PID tuning. The PID corrector loop and trajectory generation loop default rates have been optimized to provide the highest level of precision. In most applications the critical control loop is the PID corrector since it has the most significant impact on positioning performance. Because of this with default values, the PID loop is updated 4 times (8/2) during each profiler cycle to improve profile execution and minimize following errors.

The Feed-Forward gain generates a voltage output to the driver that is directly proportional to the input. The purpose of this gain is to generate a movement of the positioner as close as possible to the desired move that is independent of the encoder feedback loop. Adding this Feed-Forward gain can help reduce any encountered following errors and thus requires less compensation by the PID gain corrector. For example, if a driver and positioner respond to a constant voltage by moving at a constant speed, then feed forward input would be dictated by the SetpointSpeed.

The XPS stores standard Newport stage configuration files that can be used to quickly and easily develop the stage and system initialization (.ini) files. Below is an example of a typical stage and the type of DriverName, MotorDriverInterface and CorrectorType each is assigned. These standard Newport settings will be optimal for virtually every application and users would only need to modify their corrector loop parameters (Kp, Kd, Ki) to optimize positioner performance. Similar configurations can be adopted for non-Newport stages that are of similar motor driver types.

- Stages with high current (> 3 A) DC motor (RV, IMS) (with tachometer or back-emf estimation):

  DriverName: XPS-DRV01, 03

  - ±10 V Input gives ±ScalingVelocity (stage velocity).
  - Speed loop & Current loop configured by hardware.

  MotorDriverInterface: AnalogVelocity

  CorrectorType: PIDFFVelocity for Speed loop and PIDFFAcceleration for current loop.

- Stages with DC motor driven through a current loop (RGV) (no tachometer):

  DriverName: XPS-DRV02

  - ±10 V Input gives ±ScalingAcceleration (stage acceleration).
  - Current loop configured by hardware.

  MotorDriverInterface: AnalogAcceleration

  CorrectorType: PIDFFAcceleration

- Stages with low current (< 3 A) DC motor & tachometer (VP):

  DriverName: XPS-DRV01 in velocity mode.

  - Input 1: ±10 V results in ±ScalingVelocity (theoretical stage velocity).
  - Input 2: ±10 V results in ±ScalingCurrent (3 A).
  - Speed loop programmable.

  MotorDriverInterface: AnalogVelocity

  CorrectorType: PIDFFVelocity

- Stages with low current (<3 A) DC motor, without tachometer (ILSCC type):

  DriverName: XPS-DRV01 in voltage mode.

  - Input 1: ±10 V results in ±ScalingVoltage (48 V).

◇ Input 2: ±10 V results in ±ScalingCurrent (3 A).

MotorDriverInterface: AnalogVoltage

CorrectorType: PIDDualFFVoltage

- Stages with Stepper motor & Encoder (UTSPP, RVPE, ILSPP…):

  DriverName: XPS-DRV01 in stepper mode.

  ◇ Input 1: ±10 V results in ±ScalingCurrent in motor winding 1.

  ◇ Input 2: ±10 V results in ±ScalingCurrent in motor winding 2.

  MotorDriverInterface: AnalogStepperPosition

  CorrectorType: PIPosition

- Stages with Stepper motor & no encoder (TRA, SR50PP, PR50PP, MFAPP):

  DriverName: XPS-DRV01 in stepper mode.

  ◇ Input 1: ±10 V results in ±ScalingCurrent in motor winding 1.

  ◇ Input 2: ±10 V results in ±ScalingCurrent in motor winding 2.

  MotorDriverInterface: AnalogStepperPosition

  CorrectorType: NoEncoderPosition

These are just examples of available positioner associations in the XPS. The flexibility of positioner associations allows many other configurations to be developed to drive non-Newport positioners or other products. Before developing other configurations, the user must be aware that the main goal of creating these associations is to match the servo loop output to the appropriate driver input as stated by the manufacturer. For instance:

- The Corrector PIPosition is used when a constant voltage applied to a driver results in a constant position of the positioner (stepper motor, piezo, electrostrictive, etc.).
- Corrector PIDFFVelocity is used when a constant voltage applied to a driver results in a constant speed of the positioner (DC motor and driver board in speed loop mode).
- Corrector PIDFFAcceleration is used when a constant voltage applied to a driver results in a constant acceleration of the positioner (DC motor and driver board in current loop mode).
- Corrector PIDDualFFVoltage is used when a constant voltage applied to a driver results in a constant voltage applied to the motor (DC motor and driver board with direct PWM command).

10.1.2 XPS PIDFF Architecture

Corrector loops PIDFFVelocity, PIDFFAcceleration and PIDFFDualVoltage all use the same architecture as the PID corrector that is detailed below. PIPosition is a simplified version of this loop that is used to provide closed loop positioning via encoder feedback to stepper motor positioners.

10.1.3 PID Corrector Architecture

The PID corrector uses the following error (SetpointPosition – EncoderPosition) as its input and applies the sum of three correction terms (Kp, Kd and Ki) to determine the output.

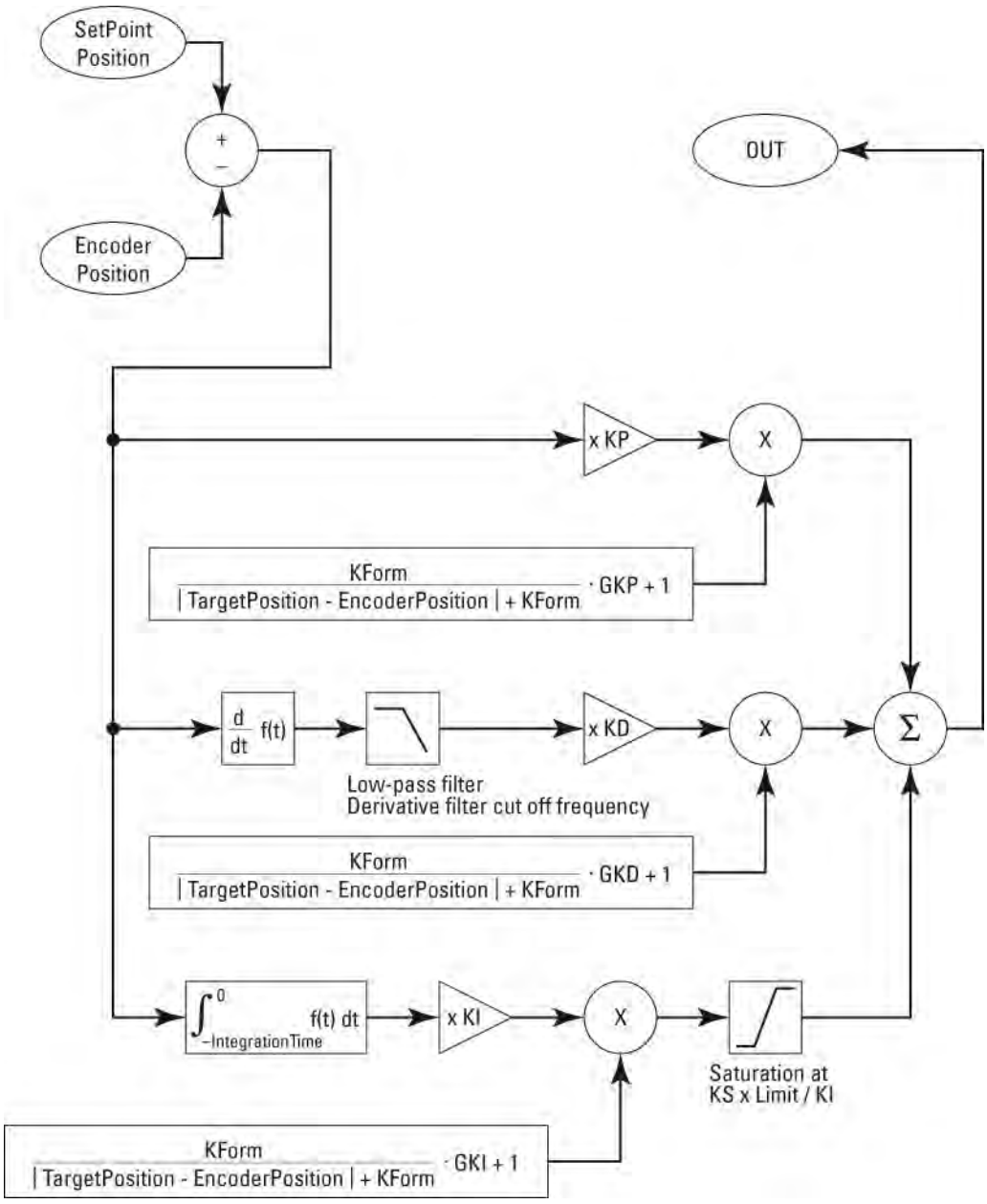


Figure 32: PID corrector architecture.

10.1.4 Proportional Term

The **Kp**, or proportional gain, multiplies the current following error of that servo cycle by the proportional gain value (Kp). The effect is to react immediately to the following error and attempt to correct it. Changes in position generally occur during commanded acceleration, deceleration, and in moves where velocity changes occur in the system dynamics during motion. As Kp is increased, the PID corrector will respond with a increased output and the error is more quickly corrected. For instance, if a positioner or group of positioners is expected to have small following errors, as is the case for small moves where overcoming static friction of the system is predominant, then the Kp may need to be increased to produce sufficient output to the driver. For larger moves, the following errors are generally larger and require lower Kp values to produce the desired

output. Also note that for larger moves the kinetic friction of the system is generally much lower than static friction and would generally require less correction gain than smaller moves. However, if Kp becomes too large, the mechanical system may begin to overshoot (encoder position > SetpointPosition), and at some point, it may begin to oscillate, becoming unstable if it does not have sufficient damping.

Kp cannot completely eliminate errors. However, since as the following error e, approaches zero, the proportional correction element, Kp x e, also approaches zero and results in some amount of steady-state error. For this reason other gain factors like Kd and Ki are required.

10.1.5 Derivative Term

The **Kd**, or derivative gain, multiplies the differential between the previous and current following error by the derivative gain value (Kd). The result of this gain is to stabilize the transient response of a system and can also be thought of as electronic damping of the Kp. The derivative acts as a gain that increases with the frequency of the variations of the following error:

$$\frac{d}{dt}=\left[\sin(2\pi\ Fr\ t)\right]=2\pi\ Fr\cos(2\pi\ Fr\ t)$$

The result is that the derived term becomes dominant at high frequencies, compared to the proportional and integral terms. For the same reason, the value of Kd is in most cases limited by high frequency resonance of the mechanics. This is why a low pass filter (cut off frequency = DerivativeFilterCutOffFrequency) is implemented in the derivative branch to limit excitation at high frequencies. Increasing the value of Kd increases the stability of the system. The steady-state error, however, is unaffected since the derivative of the steady-state error is zero.

These two gains alone can provide stable positioning and motion for the system. However to eliminate the steady state errors, an additional gain value must be used.

10.1.6 Integral Term

The Integral term **Ki** acts as a gain that increases when the frequency of the variations of the following error decrease:

$$\int\left[\sin(2\pi\ Fr\ t)\right]=-\left(\frac{1}{2}\pi\ Fr\right)\sin(2\pi\ Fr\ t)$$

The result is that the integral term becomes dominant at low frequencies, compared to the proportional and derivative terms. The gain becomes infinite when frequency = 0. Even a very small following error will generate an infinite value of the integral term. The advantage of the integral term is that it will eliminate any steady-state following error. However, the disadvantage is that the integral term can reach values where the corrector is saturated causing the system to become unstable at the end of a move and cause the positioner to hunt or dither. To reduce this effect, two additional parameters are included in the PID corrector to help prevent these instabilities, Ks and Integration Time.

Ks

The saturation limit factor Ks permits users to limit the maximum value of Ki that is applied to the total PID corrector output. The Ks saturation limit can be set between 0 and 1, a typical setting is 0.5. As an example, at a setting of 0.5, the maximum output generated by the Ki term applied to the PID output would be 0.5 x the maximum set output. However, if the Ki gain factor output is less than 0.5 x the maximum set output, then the entire gain will be applied to the PID corrector. This maximum output is set within the section MotorDriverInterface in the stages.ini using the parameters AccelerationLimit, VelocityLimit or VoltageLimit.

Integration Time

The IntegrationTime is used to adjust the duration for integration of the residual errors. This can help in applications where large following errors can occur during motion. The use of a small Integration Time value will limit the integration range to the latter parts of the move, avoiding the need of a large overshoot at the end of the move to clear the integrated following error value. The drawback is that the static error will be less compensated.

10.1.7 Variable Gains

In addition to the classical Kp, Ki, and Kd gain parameters, the XPS PID Corrector Loop also includes variable gain factors GKp, GKd, and GKi. These can be used to reduce settling time on systems that have nonlinear behavior or to tighten the control loop during the final segment of a move. For example, a positioner or stage with a high level of friction will have a response which is dependent on the size of the move: friction is negligible for a large move but becomes a predominant factor for small moves. For this reason, the required response of the system to reach the commanded position is not the same for small and large moves. The optimum value of PID parameters for small moves is very often higher than the optimum value for large moves. It is advantageous to modify PID settings depending on the move size. For users that do not need to make PID corrector adjustments (or prefer not to) benefit from the compensations provided by the variable gain correctors. This compensation is made automatically by the XPS variable gain corrector by applying a gain that is driven by the distance between the Target Position (position that must be reached at the end of the motion) and the Encoder Position. As shown in the figure below, when the distance to move completion is large, the total output gain from these parameters is fractional (the "Kform term" is fractional), but as the move size or distance to final position is small the Kform term approaches 1 and full GKx output is provided.

GKp = 10 Kp = 2
Target Position = 0
Encoder Position = -100 to 100

$$Kp_{(Kform,\ Encoder\ Position)} = \left[1 + GK \cdot \left(\frac{Kform}{|\ Target\ Position - Encoder\ Position\ | + Kform} \right) \right] \cdot Kp$$

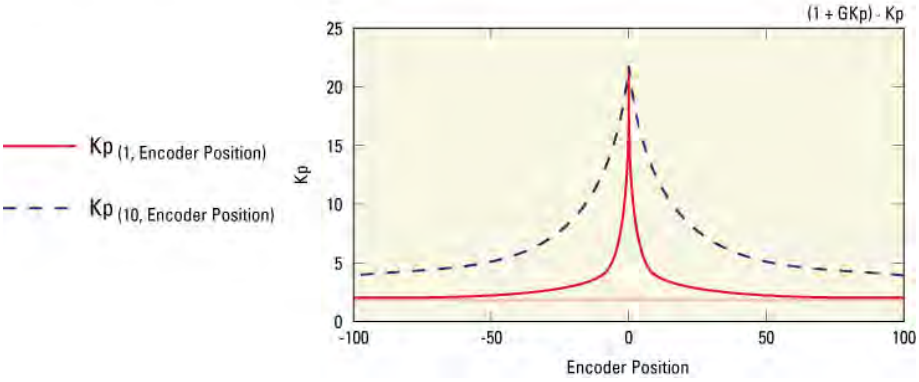


Figure 33: Variable gains.

The parameter GKx is used to adjust the amplitude of the total output and the parameter Kform is used set how soon this Gkx is applied. As seen in the figure below, if a Kform of 1 is implemented, the GKx is not applied until the positioner is very close to its target position, in this case 0. But a Kform of 10 will implement the GKx much sooner and tighten the control of the loop further from the target position. This can be very effective when positioning high inertial loads or when very short settling times are critical. The default setting for the Kform parameter is 0 for all standard Newport stages.

10.2 Filtering and Limitation

In addition to the various PID correctors and calculations, filtering and limitation parameters also have the same structure for all the correctors (PIDFFVelocity, PIDFFAcceleration and PIDFFDualVoltage, etc).

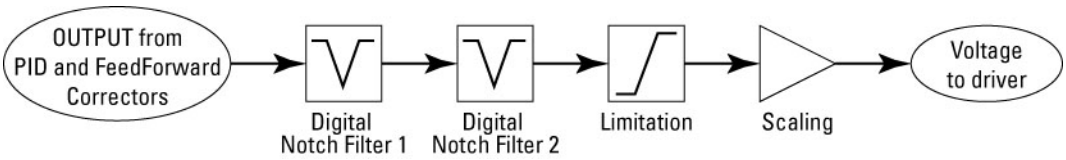


Figure 34: Filtering and limitation.

The first section of the above diagram shows the succession of two digital notch filters. Each filter is defined by its central frequency (NotchFrequency), its bandwidth (NotchBandwidth) and its gain (NotchGain).

The gain, usually in the range of 0.01 to 0.1, is the value of the amplification of a signal at a frequency equal to the central frequency and the bandwidth is the range about the central frequency for which this gain is equal to a -3 db reduction.

Notch filters are typically used to avoid the instability of the servo loop due to the mechanic's natural frequencies, by lowering the gain at these frequencies. When they are implemented, these filters add some phase shift to the signal. This phase shift increases with the filter bandwidth and must remain small in the frequency range where the servo loop is active to maintain stability. The result is that notch filters are only effective at avoiding instabilities due to excessive and constant natural frequencies.

The last section of the diagram shows the limitation and scaling features. Scaling is used to transform units of position, speed or acceleration to a corresponding voltage. The Limitation factor is a safety that is used to limit the maximum voltage that can be applied to the driver to protect against any runaway or saturation situations that may occur.

10.2.1 Current velocity and current acceleration

In XPS controller, the current velocity and current acceleration are calculated from the current position by successive derivative calculations. Because of derivative calculations these values are noisy and must be filtered by a low-pass filter to become exploitable.

Current velocity and acceleration filter parameters (stages.ini):

- Current velocity cut-off frequency: CurrentVelocityCutOffFrequency
- Current acceleration cut-off frequency: CurrentAccelerationCutOffFrequency

The CurrentVelocityCutOffFrequency (Hz) and CurrentAccelerationCutOffFrequency (Hz), set the cut-off frequencies for the low-pass filters that are applied to the CurrentVelocity and CurrentAcceleration. This filter reduces the derivative noises. They must be greater than zero (filter disabled) and less than half of the servo loop frequency (1/CorrectorISRPeriod (see system.ref)). The default value is 100 Hz which is about five times greater than the bandwidth of the position servo loop of a typical screw driven stage.

10.3 Feed Forward Loops and Servo Tuning

10.3.1 Corrector = PIDFFVelocity

The PIDFFVelocity corrector should be implemented into applications where the positioner driver requires a “speed” input (constant voltage to the driver provides constant speed output to the positioner), using MotorDriverInterface = AnalogVelocity.

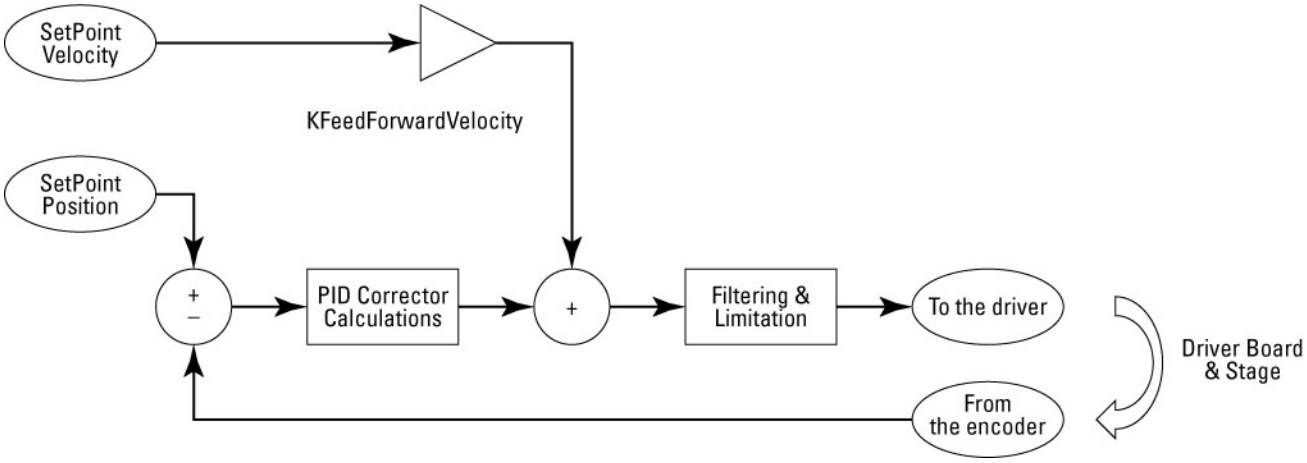


Figure 35: Corrector = PIDFFVelocity.

10.3.2 Parameters

FeedForward Method:

- Velocity
- KFeedForwardVelocity is a gain that can be applied to this feed forward.
- When the system is used in open loop, the PID output is not applied and the feed forward gain is set to 1 (the entire output of the controller is FF gain).

PID corrector:

- Total output of the PID is a speed (units/s), so:

  Kp is given in 1/s.

  Ki is given in $1/s^2$.

  Kd has no unit.

Filtering and Limitation:

- ScalingVelocity (units/s) is the theoretical speed resulting from a 10 V input to the driver.
- VelocityLimit (units/s) is the maximum speed that can be commanded to the driver.

10.3.3 Basics

For a “perfect system” (no friction, all performance factors known, no following errors), a KFeedForwardVelocity value of 1 will generate the exact amount of output required to reach the TargetPosition.

The Kd parameter is generally redundant when using the speed loop of the driver and is usually set to zero, but a higher value can be used to improve the “tightness” of the speed loop.

The proportional gain Kp drives the cut-off frequency of the closed loop.

Due to the integration of the speed command in a position by the encoder, the overall gain of the proportional path at a given frequency Frq is equal to $Kp/2\pi Frq$. This gain is equal to 1 at Frq $P = Kp/2\pi$ (close to the cut-off frequency).

This frequency must remain lower than the cut-off frequency of the speed loop of the driver and lower than the mechanic's natural frequencies to maintain stability.

The integral gain Ki drives the capability of the closed loop to overcome perturbations and to limit static error.

Due to the integration of the speed command in a position by the stage encoder, the overall gain of the integral path at a given frequency Frq is:

$$Gain = \frac{Ki}{(2 \cdot \pi \cdot Frq)^2}$$

This gain is equal to one at FrqI:

$$FrqI = \frac{1}{2 \cdot \pi} \cdot \sqrt{Ki}$$

This frequency FrqI must typically remain lower than the frequency FrqP of the proportional path to keep the stability of the servo loop.

10.3.4 Methodology of Tuning PID's for PIDFFVelocity Corrector (DC motors with or without tachometer)

1. Verify the speed in open loop (adjustment done using ScalingVelocity).
2. Close the loop, set Kp, increase it to minimize following errors to the level until oscillations/vibrations start during motion, then decrease Kp slightly to cancel these oscillations.
3. Set Ki, increase it to limit static errors and improve settling time until the appearance of overshoot or oscillation conditions. Then reduce Ki slightly to eliminate these oscillations.
4. Kd is generally not needed but it can help in certain cases to improve the response when the speed loop of the driver board is not efficient enough.

NOTE

To set the corrector parameters (loop type, Ki, Kp, Kd,…), use the following functions:

CorrectorType = PIDFFVelocity : PositionerCorrectorPIDFFVelocitySet(…)

CorrectorType = PIDFFAcceleration: PositionerCorrectorPIDFFAccelerationSet(…)

CorrectorType = PIDDualFFVoltage: PositionerCorrectorPIDDualFFVoltageSet(…)

CorrectorType = PIPosition: PositionerCorrectorPIPositionSet(…)"

10.3.5 Corrector = PIDFFAcceleration

The PIDFFAcceleration must be used in association with a driver having a torque input (constant voltage gives constant acceleration), using MotorDriverInterface = AnalogAcceleration. (AnalogSin60Acceleration, AnalogSin90Acceleration, AnalogSin120Acceleration, AnalogDualSin60Acceleration, AnalogDualSin90Acceleration or AnalogDualSin120Acceleration).

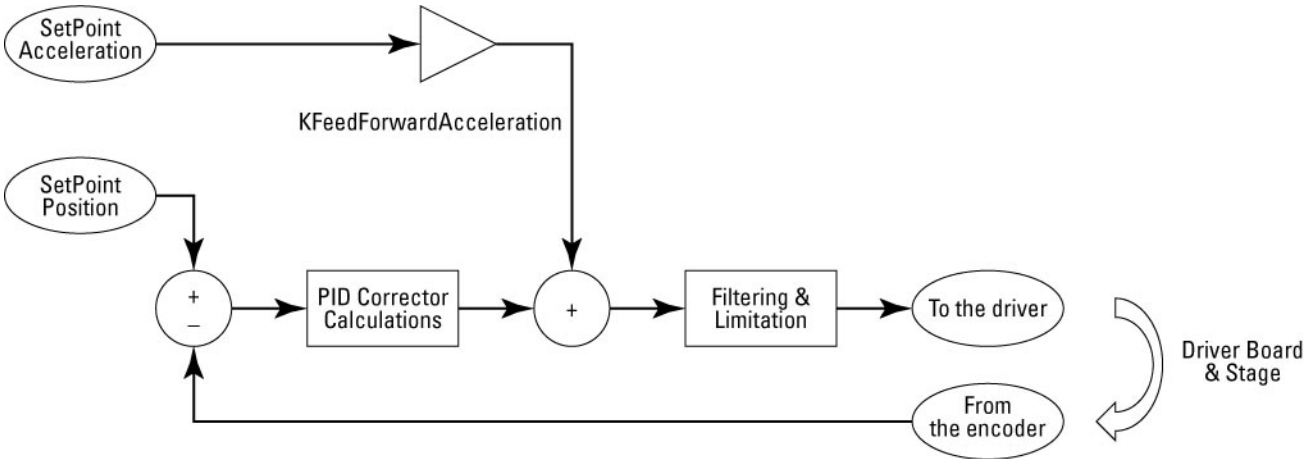


Figure 36: Corrector = PIDFFAcceleration.

10.3.6 Parameters

FeedForward method:

- A feed forward in acceleration is used.
- KFeedForwardAcceleration is a gain that can be applied to this feed forward.
- When the system is used in open loop, the PID output is cut and the feed forward gain is set to 1.

PID corrector:

- Output of the PID is an acceleration value in units/s^2.

  Kp is given in 1/s^2.

  Ki is given in 1/s^3.

  Kd is given in 1/s.

Filtering and Limitation:

- ScalingAcceleration (units/s^2) is the theoretical acceleration of the stage resulting from a 10 V input to the driver (depends on the stage payload).
- AccelerationLimit (units/s^2) is the maximum acceleration allowed to be commanded to the driver.

10.3.7 Basics

The derivative term Kd drives the cut-off frequency of the closed loop and must be adjusted first (the loop will not be stable with only Kp).

Due to the double integration of the acceleration command in a position by the stage encoder, the overall gain of the derivative path at a given frequency Frq is equal to $Kd/2\pi Frq$. This gain is equal to one at $FrqD = Kd/2\pi$ (close to servo loop cut-off frequency). This frequency must remain lower than the cut-off frequency of the current loop of the driver and lower to mechanical natural frequencies to keep the stability.

The proportional gain Kp drives mainly the capability of the closed loop to overcome perturbations at medium frequencies and to limit following errors. Due to the double integration of the acceleration command in a position by the stage encoder, the overall gain of the proportional part at a given frequency Frq is:

$$Gain = \frac{Kp}{(2 \cdot \pi \cdot Frq)^2}$$

This gain is equal to one at FrqP:

$$FrqP = \frac{1}{2 \cdot \pi} \cdot \sqrt{Kp}$$

This frequency FrqP must remain lower than the frequency FrqD of the derivative part to keep the stability.

The integral gain Ki drives the capability of the closed loop to overcome perturbations at low frequencies and to limit static error.

Due to the double integration of the acceleration command in a position by the stage encoder, the overall gain of the integral part at a given frequency Frq is:

$$Gain = \frac{Ki}{(2 \cdot \pi \cdot Frq)^3}$$

This gain is equal to one at FrqI:

$$FrqI = \frac{1}{2 \cdot \pi} \cdot Ki^{\frac{1}{3}}$$

This frequency FrqI must remain lower than the frequency FrqP of the proportional part to keep the stability.

10.3.8 Methodology of Tuning PID's for PIDFFAcceleration Corrector (direct drive DC motors)

1. Verify the AccelerationFeedForward in open loop (adjustment done using ScalingAcceleration).
2. Close the loop, set Kd, increase it to minimize following errors until vibrations appear during motion.
3. Decrease Kd to eliminate oscillations.
4. Set Kp, increase it to minimize following errors until the appearance of oscillations, decrease it to eliminate oscillations.
5. Set Ki, increase it to limit static errors and settling time until the appearance of overshoot/oscillations.

NOTE

To set the corrector parameters (loop type, Ki, Kp, Kd,…), use the following functions:

CorrectorType = PIDFFVelocity : PositionerCorrectorPIDFFVelocitySet(…)

CorrectorType = PIDFFAcceleration: PositionerCorrectorPIDFFAccelerationSet(…)

CorrectorType = PIDDualFFVoltage: PositionerCorrectorPIDDualFFVoltageSet(…)

CorrectorType = PIPosition: PositionerCorrectorPIPositionSet(…)"

10.3.9 Corrector = PIDDual FFVoltage

The PIDDualFFVoltage must be used in association with a driver having a voltage input (constant voltage gives constant motor voltage), using MotorDriverInterface = AnalogVoltage.

Can also be used in velocity or acceleration command.

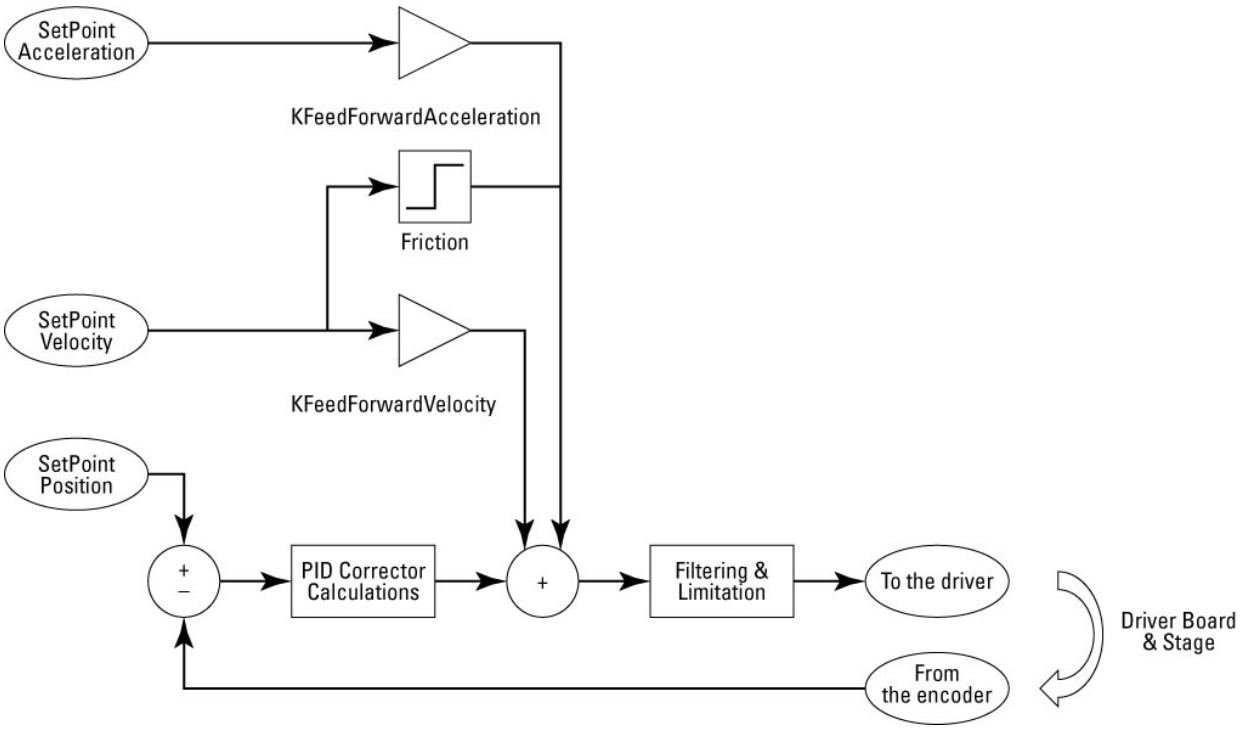


Figure 37: Corrector = PIDDual FFVoltage.

10.3.10 Parameters

FeedForward method:

- 3 feed forwards are used: Speed, Acceleration and Friction.
- KFeedForwardAcceleration is a gain that can be applied to the feed forward in acceleration.
- KFeedForwardVelocity is a gain that can be applied to the feed forward in velocity.
- Friction is a value which is applied with the sign of the velocity.
- When the system is used in open loop, the PID output is cut and only one feed forward in velocity is applied with the gain defined by KFeedForwardVelocityOpenLoop.

PID corrector:

- Output of the PID is a voltage.

  Kp is given in V/unit.

  Ki is given in V/unit/s.

  Kd is given in V/s/unit.

Filtering and Limitation:

- ScalingVoltage is the theoretical motor voltage resulting from a 10 V input on the driver (48 V).
- VoltageLimit (volts) is the maximum motor voltage allowed to be commanded to the driver.

Refer to **Configuration Manual** for a detailed explanation.

10.3.11 Basics

The PIDDualFFVoltage corrector can be seen as a mix between the PIDFFVelocity and PIDFFAcceleration correctors. It is difficult to give a precise picture of this behavior which depends a lot on the response of the stage (speed and acceleration versus motor voltage).

10.3.12 Methodology of Tuning PID's for PIDDualFF Corrector (DC motors with tachometers)

1. Adjust KFeedForwardVelocityOpenLoop to optimize the fidelity of the speed at high speed.
2. Close the loop using the same value for KFeedForwardVelocity, set Kp, increase it to minimize following errors until oscillations/vibrations appears during motion, decrease Kp to eliminate oscillations.
3. Set Kd, increase until oscillations/vibrations appear during motion, and decrease it to eliminate oscillations.
4. Increase Ki to cancel static error and minimize settling time until appearance of overshoot/oscillations.

10.3.13 Corrector = PIPosition

PIPosition corrector can be used with AnalogStepperPosition or AnalogPosition interface.

The AnalogPosition interface is to be used with a driver having a position input (example = piezo driver).

The AnalogStepperPosition interface is to be used with a driver having two sine and cosine current inputs (constant voltage gives constant currents in motor windings so position is constant).

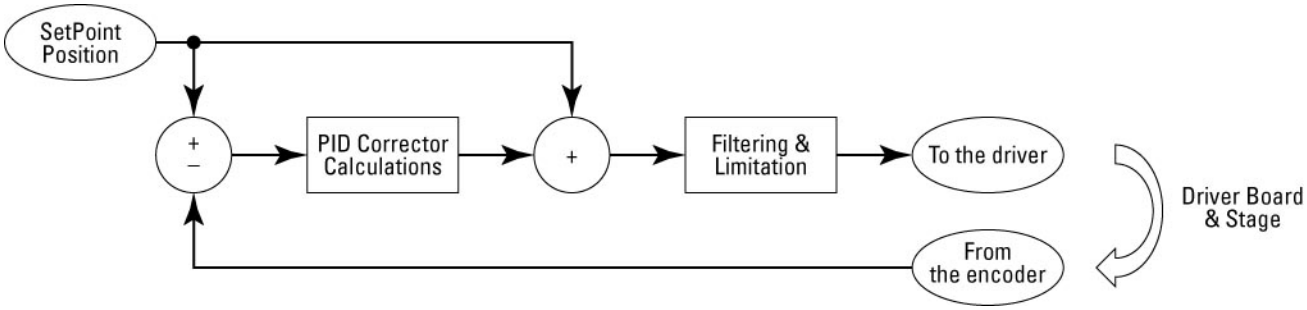


Figure 38: Corrector = PIPosition.

10.3.14 Parameters

FeedForward:

- One feed forward in position. No adjustable gain.
- When the system is used in open loop, the PI output is cut and the feed forward in position is applied.

PI corrector:

- Output of the PI is a position.

  Kp has no units.

  Ki is given in 1/s.

10.3.15 Basics & Tuning

In most cases, only Ki is needed to correct static errors.

The overall gain of the integral part of the servo loop at a given frequency Frq is:

$$\mathbf{Gain} = \frac{\mathbf{Ki}}{2 \cdot \pi \cdot \mathbf{Frq}}$$

This gain is equal to one at:

$$\mathbf{FrqI} = \frac{\mathbf{Ki}}{2 \cdot \pi}$$

11.0 Analog Encoder Calibration

This section refers only to analog sine encoder inputs. The purpose of the analog encoder interpolation feature is to improve the stage accuracy by detecting and correcting analog encoder errors such as offsets and sine to cosine amplitude differences.

Other kinds of errors can exist in the encoder such as impure sine or cosine signals. This feature will not compensate for them and will disturb the results of the calibration process.

Also, this calibration process assumes that the errors are small, i.e., less than a few percent.

Below are figures and numbers to illustrate the type of errors and their impact on accuracy.

11.1 Analog Encoder Errors

Offset Error

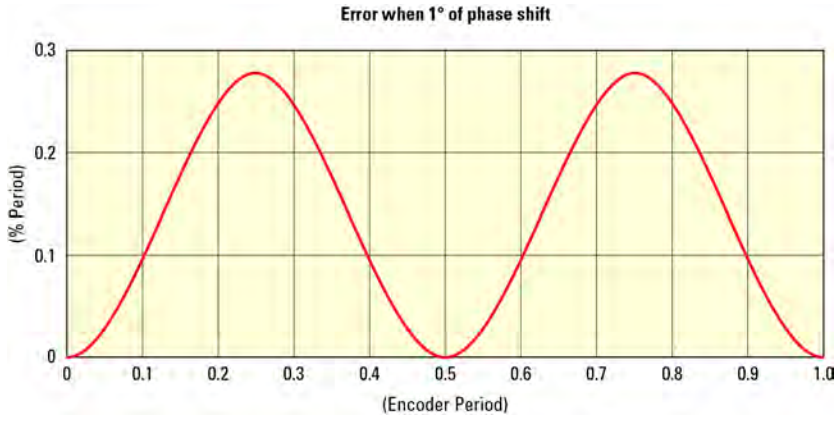


Figure 39: Offset error.

The offset error generates 0.32% interpolation error per percent offset on the sine or cosine signals. With a 20 µm scale pitch, 1% sine offset generates 63.5 nm peak to peak interpolation error.

NOTE

The real signal is not always symmetrical to 0. The offset error is defined as the difference between the signal's horizontal axis where it is symmetrical and 0.

Amplitude Mismatch

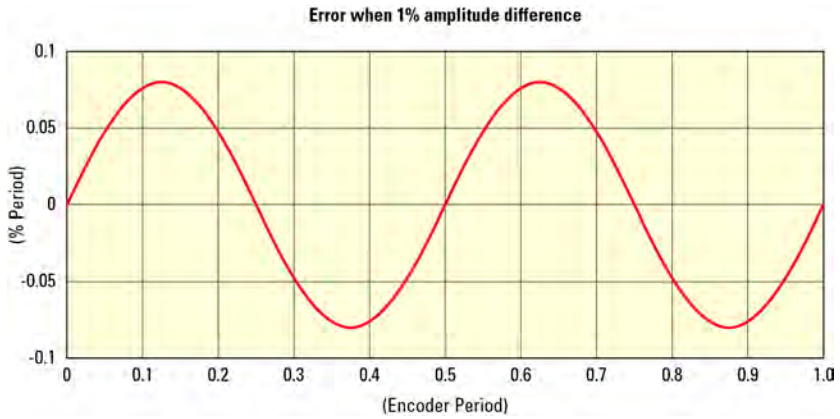


Figure 40: Amplitude mismatch.

The amplitude mismatch between sine and cosine signals generates 0.17% interpolation error per percent amplitude mismatch. With a 20 µm scale pitch, 1% amplitude mismatch generates 33 nm peak to peak interpolation error.

NOTE

Positive amplitude is the distance between the signal's maximum value and the signal axis. Negative amplitude is the distance between the signal's minimum value and the signal axis. If the positive amplitude and negative amplitude are not equal, there is amplitude mismatch.

Combined Errors

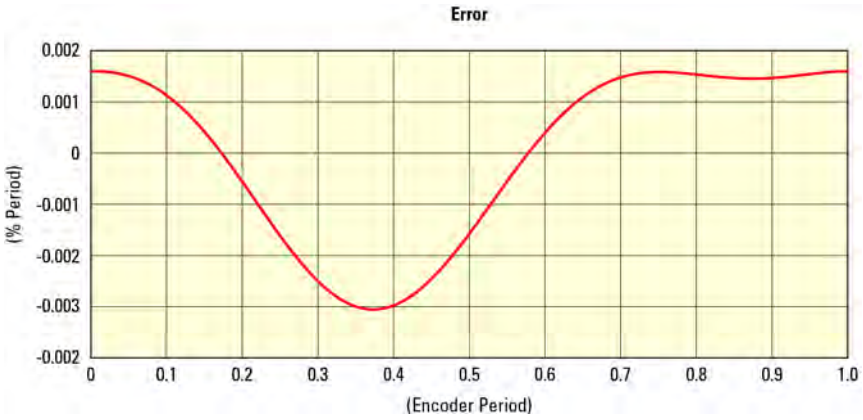


Figure 41: Combined errors.

The combination of these errors is not a simple sum but is more likely a root mean square relationship. With a 20 µm scale pitch, 1% sine offset, 1% cosine offset, 1% amplitude mismatch between sine and cosine generates 93.37 nm peak to peak error.

$$20000\sqrt{(0.32\%)^2+(0.32\%)^2+(0.164\%)^2}=96.27$$

Note that the calculated value, 96.27 nm is different than the measured 93.37 nm.

11.2 Analog Encoder Compensation Feature

The compensation for repeatable distortions of the analog encoder input signals is always active. It uses the following parameters read from the stages.ini file. The default values are 0 for all stages:

EncoderSinusOffset = 0 Volts

EncoderCosinusOffset = 0 Volts

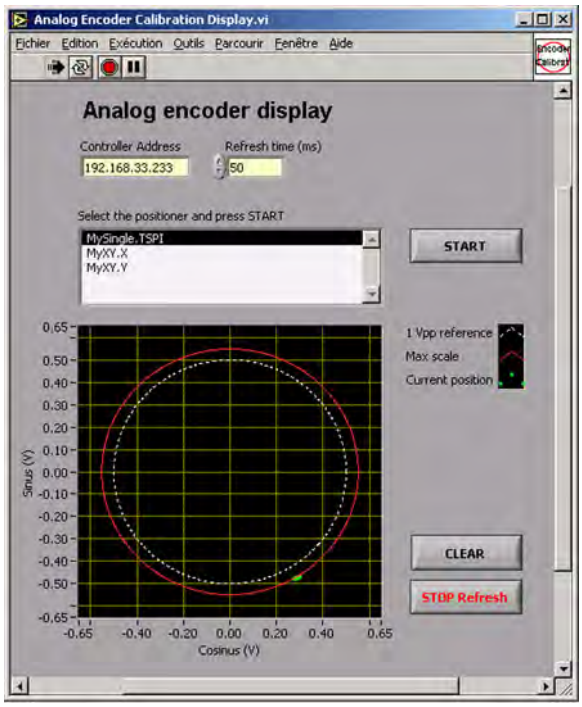
EncoderDifferentialGain = 0

EncoderPhaseCompensation = 0 deg

The function **GroupInitializeWithEncoderCalibration**() initializes the positioner and runs the encoder calibration process. During calibration, the stage moves for 25 EncoderScalePitch and the controller determines the appropriate calibration values. The controller though, will not automatically apply these values.

The function **PositionerEncoderCalibrationParametersGet**() returns the results of the last encoder calibration. To apply these values, add them manually to the appropriate section in the stages.ini file, and reboot the controller.

Stored on the XPS controller (accessible through the XPS webpage **Documentation -> Drivers & Examples -> labview -> XPS-Q8 Controller -> Examples**), embedded in Examples.llb, there is a LabVIEW application to display the current analog encoder values. The display zone matches the maximum possible amplitude of the analog signals. When they are larger than this, the AD converter will clip and the interpolation error will increase dramatically. The dotted circle represents the 1 Volt peak to peak "ideal" encoder, the red circle represents the current mean encoder settings and the green dot the current encoder value. This application uses the function PositionerEncoderAmplitudeValuesGet() for display.



Example of the use of the functions

GroupInitializeWithEncoderCalibration(MyGroup)

PositionerEncoderCalibrationParametersGet(MyGroup.MyStage)

This function returns the encoder calibration parameter values: encoder sine signal offset, encoder cosine signal offset, encoder differential gain, and encoder phase compensation. These values need to be entered in the appropriate section of the stages.ini.

PositionerEncoderAmplitudeValuesGet(MyGroup.MyStage)

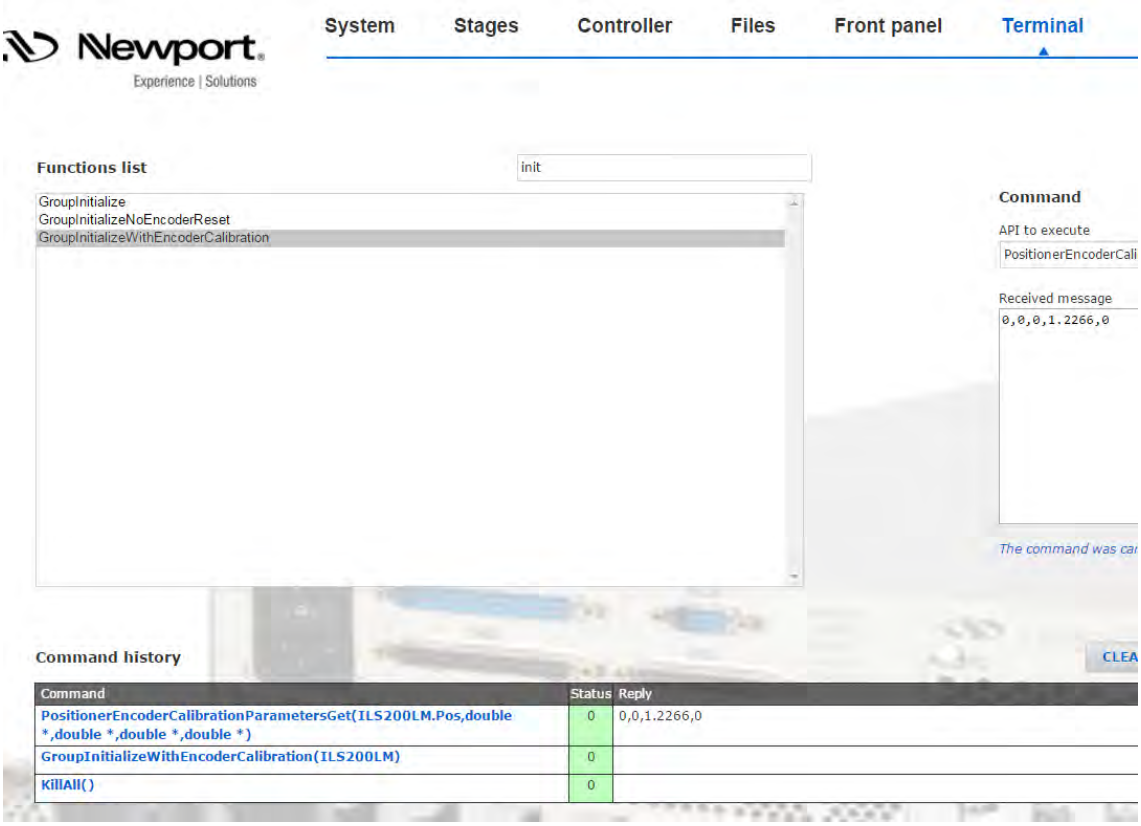
This function returns the encoder amplitude values: encoder sine signal maximum amplitude value, encoder sine signal current amplitude value, encoder cosine signal maximum amplitude value and encoder cosine signal current amplitude value.

Following is the complete process for calibrating a stage with an analog encoder interface:

11.3 Calibration Procedure

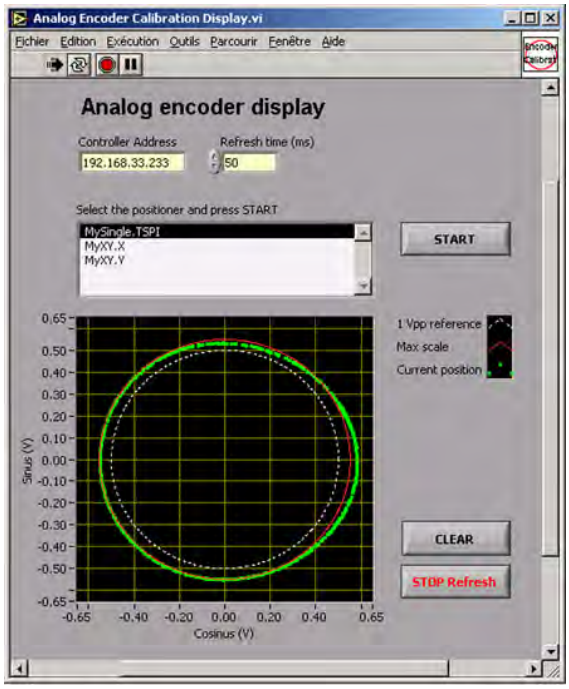
Step 1

Initialize the positioner, run the calibration routine and read the encoder calibration parameters.



Step 2

Start the AnalogEncoderCalibrationDiplay VI which is found in the ftp site. Move the positioner at very low speed.

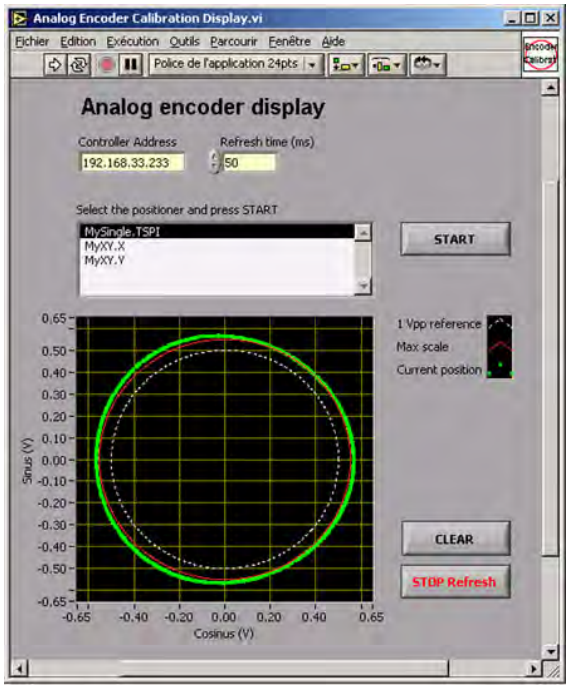


Notice the variations between the actual (green) values and the ideal (red) values. In this case, it makes sense to apply new compensation values.

Step 3

Apply the compensation values gathered in step 1 into the stages.ini; reboot the controller.

Initialize the positioner: run the AnalogEncoderCalibrationDiplay VI; move the positioner at a very low speed.



Notice the difference to the previous results. It might be necessary to run the compensation at several positions and several times to optimize the results.

12.0 Excitation Signal

12.1 Introduction

The excitation-signal function generates a typical signal (a sine, a blank noise or an echelon signal) that the controller sends to motors to excite the system. In measuring the output signal of the excited system, we can determine some system characteristics, such as the system transfer function.

12.2 How to Use the Excitation-Signal Function

The PID excitation-signal function is only available with the stages controlled in acceleration (acceleration control, ex: brushless/linear motors), velocity (velocity control) or in voltage (voltage control). It is not used with the stages controlled in position (ex: stepper motors).

The excitation-signal function **PositionerExcitationSignalSet** can be executed only when the positioner is in the "READY" state. When the excitation-signal function is in process, the positioner is in the "ExcitationSignal" state. At the end of the process, the positioner returns to the "READY" state (see **2.2 State Diagrams**).

This function sends an excitation command to the motor over a time period. This function is allowed for "PIDFFAcceleration", "PIDFFVelocity" or "PIDDualFFVoltage" control loop. The parameters to configure are *signal type* (0:sine, 1:echelon,2:random-amplitude,3:random-pulse-width binary-amplitude, integer), *frequency* (Hz, double), *amplitude* (acceleration, velocity or voltage unit, double) and *during time* (seconds, double).

The effective functional parameters for each mode are: (Limit means AccelerationLimit, VelocityLimit or VoltageLimit) :

– Sine signal mode : *Frequency* (>=1 and <= 5000), *Amplitude* (>0 and <= Limit), *Time*(>0)

– Echelon signal mode : *Amplitude* (>0 and <= Limit, or <0 and >= -Limit), *Time* (>0).

  + During *Time* : Signal = *Amplitude*

  + End of *Time* : Signal = 0

– Random-amplitude signal mode : *Amplitude* (>0 and <= Limit), *Time*(>0), *Frequency* (>= 1 and <= 5000).

  The signal is generated with a random value at every controller base time (Tbase = 0.1 ms), then is filtered with a second order low-pass filter at the cut-off *Frequency* value.

– Random-pulse-width binary-amplitude signal mode :

  Amplitude (>0 and ≤ Limit), *Time* (>0), *Frequency* (≥1 and ≤5000).

  The signal is a sequence of pulses (Signal = *Amplitude* or = 0) with the pulse randomly varied in width (multiple of Tbase).

  Frequency is the controlled system band-width (*cut-off frequency*), necessary for the PRBS (*Pseudo Random Binary Sequence*) function configuration.

The non-effective functional parameters can accept any value, the value 0 is recommended for simplicity.

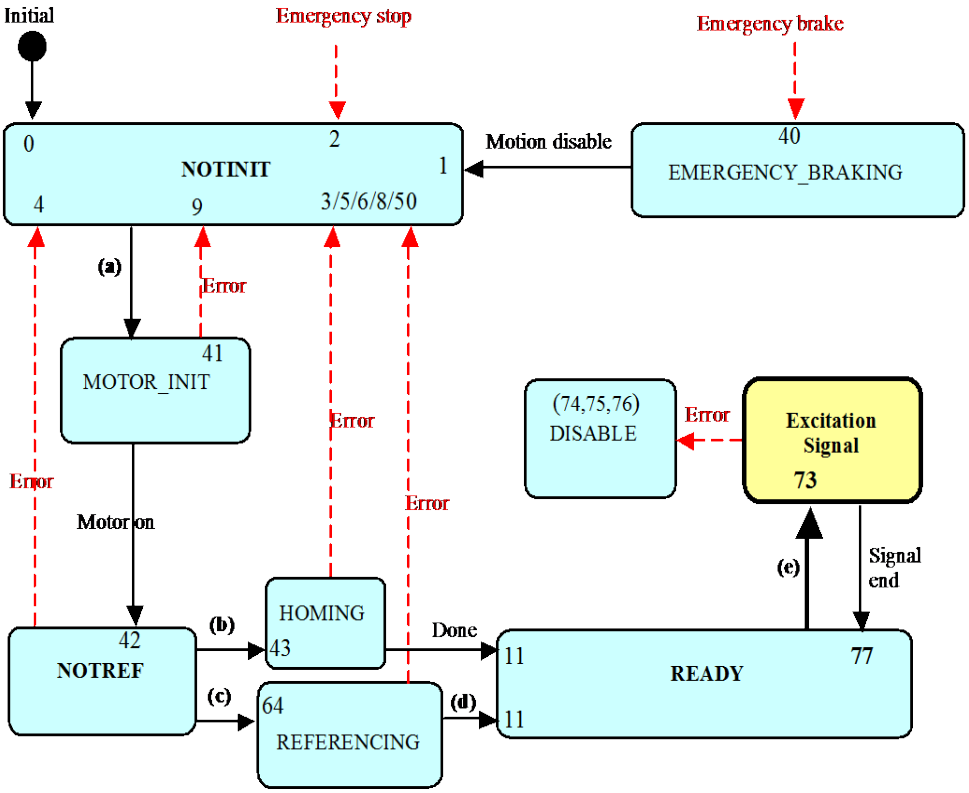
The *PositionerExcitationSignalGet()* function is used to get the parameters previously used with the *PositionerExcitationSignalSet()* function.

12.3 Group State Diagram

Notes :

The numbers in the boxes represent the values of the group status.

Bold transitions are driven by function, the others are internal transitions.



(a) GroupInitialize

(b) GroupHomeSearch or **(c)** GroupReferencingStart and **(d)** GroupReferencingStop

(e) PositionerExcitationSignalSet

12.4 Function Description

- **PositionerCorrectorExcitationSignalGainGet()**
- **PositionerCorrectorExcitationSignalGainSet()**

(see **Programmer's Manual**)

13.0 Introduction to XPS Programming

For advanced applications and repeating tasks, it is usually better to sequence different functions in a program rather than executing them manually via the web site interface. Motion programs can be written in different ways, but essentially are distinguished between host-PC-managed and XPS-managed processes. A host-PC-managed program uses the Ethernet TCP/IP interface from a PC to control the XPS. The XPS-managed process is controlled internally by the XPS controller via a TCL script.

The chapter provides a brief introduction of the different ways of programming the XPS. This section, however, cannot address all details. For further information, refer to the software drivers manual of the XPS controller which are accessible via the XPS web site.

Host-managed processes

Host-managed processes are recommended for applications that require a lot of data management or a lot of digital communication with devices other than the XPS controller. In this case, it is more efficient to control the process from a dedicated program that runs on a PC and which sends (and receives) information to (and from) the XPS controller via the Ethernet TCP/IP communication interface. For more details, please refer to the Software .NET Manual.

XPS-managed processes (TCL)

The XPS controller is also capable of controlling processes directly using TCL scripts. TCL stands for Tool Command Language and is an open-sourced, object oriented, command language. With only a few fundamental constructs, it is very easy to learn and it is almost as powerful as C. Users of the XPS can use TCL to write complete application code with any function. The TCL script can be executed in real time but in the background, thus utilizing time that the controller does not need for servo or communication. Multiple TCL programs can run in a time sharing mode. To learn more about implementing TCL, refer to the TCL website www.tcl.tk.

The advantages of XPS-managed processes compared to host-managed processes are faster execution and better synchronization in many cases without any processing time taken by the communication link. XPS-managed processes or sub-processes are particularly valuable for repeating tasks, tasks that run in a continuous loop, and tasks that require a lot of data from the XPS controller. Examples include: anti-collision processes (processes that utilize security switches to stop motion when stages are in danger of collision); tracking, auto-focusing or alignment processes (processes that use external data inputs to control the motion); or custom initialization routines (processes that must constantly be executed during a system's use).

The XPS controller has real-time multi-tasking functionality, and with most applications there is not only a choice between a host-managed or an XPS-managed process, but also a recognition of splitting the application into the right number of sub-tasks, and defining the most efficient process for each sub-task. An efficient process design is one of the main challenges with complex and critical applications in terms of time and precision. It is recommended to spend time thinking about the proper process definition and the best approach to control the XPS using a program.

However, not all details can be addressed in this chapter.

NOTE

The controller's time and date should be used only as reference. It is not intended to be used as an absolute reference.

13.1 TCL Generator

The TCL generator provides a convenient way of generating simple executable TCL scripts. These scripts are also a good place to start for the development of more complex scripts. Note that applications that are memory intensive or require links other XPS may require a script that is external to the XPS.

The TCL generator is accessible from the terminal page of the XPS web site. Clicking the GENERATE TCL button generates a TCL script that includes the commands previously executed and listed in the Command history list. Note that the command order in the generated TCL script is chronological hence the order is the inverse of the Command history list order. The name of the generated TCL script can be specified after clicking GENERATE TCL otherwise a default name is given, New history yyyy-mm-dd.tcl. The generated TCL script is stored in the controller and can be downloaded, edited or deleted under the webpage **Files → TCL script**.

Example

This is an example using three stages, two in an XY group (named XY) and one in a SingleAxis group (named S).

The following functions were executed in the Terminal web page.

KillAll()

GroupInitialize(S)

GroupInitialize(XY)

GroupHomeSearch(S)

GroupHomeSearch(XY)

GroupMoveAbsolute(S, 25)

GroupMoveAbsolute(S, 0)

GPIODigitalSet(GPIO3.DO, 63, 0)

EventExtendedConfigurationTriggerSet(XY.XYLineArcTrajectory.Start, 0, 0, 0, 0)

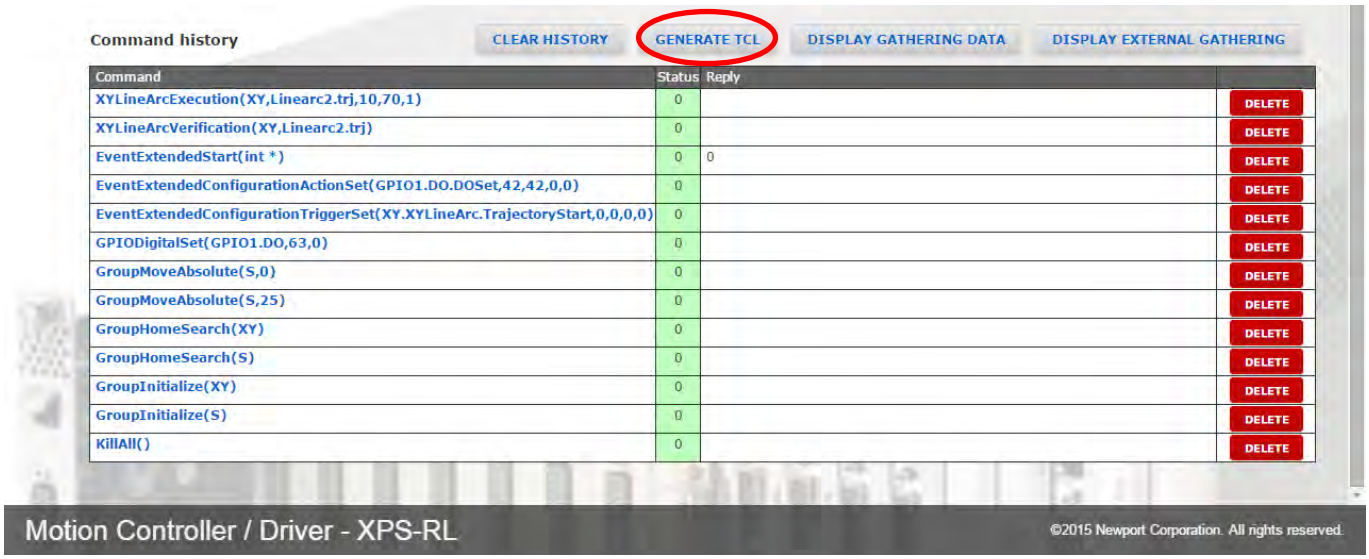
EventExtendedConfigurationActionSet(GPIO3.DO.DOSet, 42, 42, 0, 0)

EventExtendedStart()

XYLineArcVerification(XY, Linearc2.trj)

XYLineArcExecution(XY, Linearc2.trj, 10, 70, 1)

Then, the “GENERATE TCL” button is pressed to create a TCL script file. The default file name is "New history yyyy-mm-dd.tcl". When executed, that TCL file will execute all of the functions used individually in the terminal.



To execute the script, use the API function, TCLScriptExecute() and designate task name and parameter. Example with the default TCL script name: TCLScriptExecute(New history yyyy-mm-dd.tcl, task1, 0). Alternatively the user has the option to execute the TCL script and either block the socket or designate a priority level to the task.

13.1.1.1 Execute the TCL script with the given task name (in this example task1) and block the socket until the script terminates use the API TCLScriptExecureAndWait(New history yyyy-mm-dd.tcl, task1, 0, char*).

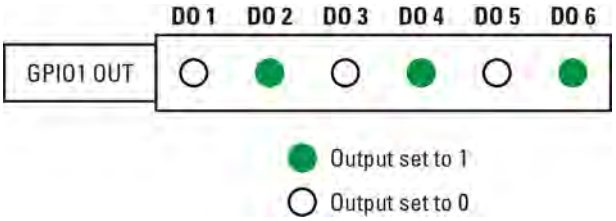
13.1.1.2 Execute the TCL script with the given task name (in this example task1) at a user defined priority level: HIGH, MEDIUM or LOW use the API TCLScriptExecuteWithPriority: (New history yyyy-mm-dd.tcl, task1,HIGH, 0).

In this example, after initializing and homing both groups, the TCL script moves the single axis stage to the position of 70 units, then to the position of –70 units. It then sets all pins 1 - 6 on the digital output GPIO3 to 0.

Once checked, the line arc trajectory defined in the Linearc2.trj file gets executed with a velocity of 10 units/s and an acceleration of 70 units/s^2. When this trajectory starts, more precisely when the positioner of the X axis starts moving, the bits #2, #4 and #6 of the output GPIO3 are set to 1 (42 = 101010).

NOTE

GPIO3.DO is only available for the Basic GPIO option.



NOTE

Selecting the function TCLScriptExecute() from the terminal menu opens a drop-down list for the available TCLFileNames. However, this list is limited to 100 entries.

To kill a running script, use the API function TCLScriptKill() and input the user defined task name; example TCLScriptKill(task1). Alternatively the user can kill all running scripts by using the API function TCLScriptKillAll(). To get a list of task names, of all running TCL scripts, use the API function TCLScriptRunningListGet().

To learn more about TCL programming, refer to the TCL website.

13.2 Running Processes in Parallel

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server both read from and write to the socket that binds the connection.

Sockets are interfaces that can “plug into” each other over a network. Once “plugged in”, the connected programs can communicate.

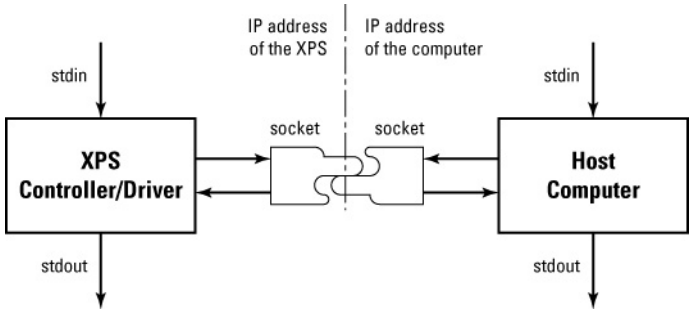


Figure 42: Running processes in parallel.

XPS uses blocking sockets. In other words, the programs/commands are “blocked” until the request for data has been satisfied. When the remote system writes data on the socket, the read operation will complete it and write the data in the received message window of the Terminal menu (‘0’ if command has been executed without error, or the error number in case of an error). That way, commands are executed sequentially since each command always waits for a response before finishing and then allowing execution of the next function. The main benefit of using this type of socket is that an execution acknowledgement is sent to the host computer with each function. In case of any error, it allows an exact diagnostic, which function has caused the error. It also allows a precise sequential process execution. On the other hand, more functions could be sent in parallel using non-blocking sockets. However, the drawback is that it is almost impossible to diagnose which function caused an error.

To execute several processes in parallel, for instance to request the current position during a motion and other data simultaneously, it is possible to communicate to the XPS controller via different sockets. The XPS controller supports a maximum number of 84 parallel opened sockets. The total number of open communication channels to the XPS controller, be it via the website, TCL scripts, a LabVIEW program, or any other program can not be larger than 84.

Users who prefer not to use blocking sockets, or whose programming languages don’t support multiple sockets, such as Visual Basic versions prior to version .Net, can disable the blocking feature by setting a low TCPTimeOut value, 20 ms for instance. In this case, the XPS will unblock the last socket after the TCPTimeOut time. However, this method loses the ability to pinpoint which commands were not properly executed.

Service Form

Your Local Representative

Tel.: _____

Fax: _____

Name: _____

Company: _____

Address: _____

Country: _____

P.O. Number: _____

Item(s) Being Returned: _____

Model#: _____

Return authorization #: _____
(Please obtain prior to return of item)

Date: _____

Phone Number: _____

Fax Number: _____

Serial #: _____

Description: _____

Reasons of return of goods (please list any specific problems): _____

Newport®

Visit Newport Online at:
www.newport.com

North America & Asia
Newport Corporation
1791 Deere Ave.
Irvine, CA 92606, USA

Sales
Tel.: (800) 222-6440
e-mail: sales@newport.com

Technical Support
Tel.: (800) 222-6440
e-mail: tech@newport.com

Service, RMAs & Returns
Tel.: (800) 222-6440
e-mail: service@newport.com

Europe
MICRO-CONTROLE Spectra-Physics S.A.S
9, rue du Bois Sauvage
91055 Évry CEDEX
France

Sales
Tel.: +33 (0)1.60.91.68.68
e-mail: france@newport.com

Technical Support
e-mail: tech_europe@newport.com

Service & Returns
Tel.: +33 (0)2.38.40.51.55

mks
Newport® Ophir® Spectra-Physics®