MCS2

Programmer's Guide

www.smaract.com

SmarAct
PERFECT MOTION

The information given in this document was carefully checked by our team and is constantly updated. Nevertheless, it is not possible to fully exclude the presence of errors. In order to always get the latest information, please contact our technical sales team.

SmarAct GmbH, Schuette-Lanz-Strasse 9, D-26135 Oldenburg
Phone: +49 (0) 441 - 800879-0, Telefax: +49 (0) 441 - 800879-21
Internet: www.smaract.com, E-Mail: info@smaract.com

Document Version: 1.0.6

TABLE OF CONTENTS

1 Introduction **10**
1.1 Terminologies 10

2 General Concepts **12**
2.1 Connecting and Disconnecting 12
2.1.1 Locators for Device Identification 12
2.1.2 Finding Devices 13
2.1.3 Network Interface Configuration 13
2.2 Properties 14
2.3 Accessing Properties 14
2.3.1 Synchronous Access 15
2.3.2 Asynchronous Access 16
2.3.3 High-Throughput Asynchronous Access 18
2.3.4 Call-and-Forget Mechanism 20
2.3.5 Request Ready Notification 21
2.4 Event Notifications 22
2.5 Positioner Types 23
2.5.1 Custom Positioner Types 23
2.6 Moving Positioners 24
2.6.1 Calibrating 25
2.6.2 Referencing 27
2.6.3 Open-Loop Movements 27
2.6.4 Closed-Loop Movements 28
2.6.5 Stopping Movements 31
2.6.6 Overwriting Movement Commands 32
2.6.7 Movement Feedback 32
2.7 Defining Positions 34
2.7.1 Reference Marks 35
2.7.2 Positioners With Single Reference Marks 36
2.7.3 Positioners With Multiple Reference Marks 38
2.7.4 Positioners With Endstop Reference 40
2.7.5 Shifting the Measuring Scale 41
2.8 State Flags 41
2.8.1 Device State Flags 41
2.8.2 Module State Flags 42
2.8.3 Channel State Flags 44
2.9 Sensor Power Modes 46
2.10 PicoScale Sensor Module 47
2.11 Following Error Detection 48
2.12 Software Range Limit 49

2.13 Stop Broadcasting 49
2.13.1 Stop Broadcast Configuration 50
2.14 Command Groups 51
2.14.1 Command Groups vs. Output Buffer 53
2.15 Trajectory Streaming 53
2.15.1 General Streaming Concept 54
2.15.2 Basic Approach 56
2.15.3 Options 56
2.15.4 Trigger Modes 57
2.15.5 Stream Events 58
2.15.6 Maximum Stream Rates 59
2.16 Auxiliary Inputs and Outputs 59
2.16.1 Digital Device Input 60
2.16.2 Fast Digital Outputs 60
2.16.3 General Purpose Digital Inputs/Outputs 60
2.16.4 Fast Analog Inputs 62
2.16.5 Using Analog Inputs as Control-Loop Feedback 63
2.16.6 Analog Outputs 64
2.17 Input Trigger 65
2.17.1 Disabled Mode 66
2.17.2 Emergency Stop Mode 66
2.17.3 Stream Sync Mode 67
2.17.4 Command Group Sync Mode 68
2.17.5 Event Trigger Mode 69
2.18 Output Trigger 70
2.18.1 Constant Mode 71
2.18.2 Position Compare Mode 71
2.18.3 Target Reached Mode 73
2.18.4 Actively Moving Mode 74
2.19 Feature Permissions 74

3 Function Reference 75
3.1 Function Summary 75
3.2 Detailed Function Description 78
3.2.1 SA_CTL_GetFullVersionString 78
3.2.2 SA_CTL_GetResultInfo 79
3.2.3 SA_CTL_GetEventInfo 80
3.2.4 SA_CTL_FindDevices 81
3.2.5 SA_CTL_Open 83
3.2.6 SA_CTL_Close 84
3.2.7 SA_CTL_Cancel 85
3.2.8 SA_CTL_GetProperty_i32 86
3.2.9 SA_CTL_SetProperty_i32 88
3.2.10 SA_CTL_SetPropertyArray_i32 89
3.2.11 SA_CTL_GetProperty_i64 90
3.2.12 SA_CTL_SetProperty_i64 91
3.2.13 SA_CTL_SetPropertyArray_i64 92
3.2.14 SA_CTL_GetProperty_s 93

3.2.15 SA_CTL_SetProperty_s ... 95
3.2.16 SA_CTL_RequestReadProperty ... 96
3.2.17 SA_CTL_ReadProperty_i32 ... 98
3.2.18 SA_CTL_ReadProperty_i64 ... 99
3.2.19 SA_CTL_ReadProperty_s ... 100
3.2.20 SA_CTL_RequestWriteProperty_i32 ... 102
3.2.21 SA_CTL_RequestWriteProperty_i64 ... 104
3.2.22 SA_CTL_RequestWriteProperty_s ... 105
3.2.23 SA_CTL_RequestWritePropertyArray_i32 ... 106
3.2.24 SA_CTL_RequestWritePropertyArray_i64 ... 107
3.2.25 SA_CTL_WaitForWrite ... 108
3.2.26 SA_CTL_CancelRequest ... 109
3.2.27 SA_CTL_CreateOutputBuffer ... 110
3.2.28 SA_CTL_FlushOutputBuffer ... 111
3.2.29 SA_CTL_CancelOutputBuffer ... 112
3.2.30 SA_CTL_OpenCommandGroup ... 113
3.2.31 SA_CTL_CloseCommandGroup ... 114
3.2.32 SA_CTL_CancelCommandGroup ... 115
3.2.33 SA_CTL_WaitForEvent ... 116
3.2.34 SA_CTL_Calibrate ... 118
3.2.35 SA_CTL_Reference ... 120
3.2.36 SA_CTL_Move ... 122
3.2.37 SA_CTL_Stop ... 124
3.2.38 SA_CTL_OpenStream ... 125
3.2.39 SA_CTL_StreamFrame ... 127
3.2.40 SA_CTL_CloseStream ... 129
3.2.41 SA_CTL_AbortStream ... 131

4 Property Reference ... 132
4.1 Property Summary ... 132
4.2 Device Properties ... 137
4.2.1 Number of Channels ... 137
4.2.2 Number of Bus Modules ... 137
4.2.3 Device State ... 138
4.2.4 Device Serial Number ... 139
4.2.5 Device Name ... 140
4.2.6 Emergency Stop Mode ... 141
4.2.7 Network Discover Mode ... 142
4.3 Module Properties ... 143
4.3.1 Power Supply Enabled ... 143
4.3.2 Module State ... 144
4.3.3 Number of Bus Module Channels ... 145
4.4 Positioner Properties ... 145
4.4.1 Amplifier Enabled ... 145
4.4.2 Positioner Control Options ... 146
4.4.3 Actuator Mode ... 147
4.4.4 Control Loop Input ... 149
4.4.5 Sensor Input Select ... 150

4.4.6 Positioner Type 151
4.4.7 Positioner Type Name 152
4.4.8 Move Mode 152
4.4.9 Channel State 154
4.4.10 Position 155
4.4.11 Target Position 156
4.4.12 Scan Position 156
4.4.13 Scan Velocity 157
4.4.14 Hold Time 158
4.4.15 Move Velocity 159
4.4.16 Move Acceleration 160
4.4.17 Max Closed Loop Frequency 161
4.4.18 Default Max Closed Loop Frequency 162
4.4.19 Step Frequency 163
4.4.20 Step Amplitude 163
4.4.21 Following Error Limit 164
4.4.22 Broadcast Stop Options 165
4.4.23 Sensor Power Mode 166
4.4.24 Sensor Power Save Delay 167
4.4.25 Position Mean Shift 168
4.4.26 Safe Direction 169
4.4.27 Control Loop Input Sensor Value 170
4.4.28 Control Loop Input Aux Value 171
4.4.29 Target To Zero Voltage Hold Threshold 172
4.5 Scale Properties 173
4.5.1 Logical Scale Offset 173
4.5.2 Logical Scale Inversion 174
4.5.3 Range Limit Min 175
4.5.4 Range Limit Max 175
4.6 Calibration Properties 176
4.6.1 Calibration Options 176
4.6.2 Signal Correction Options 177
4.7 Referencing Properties 179
4.7.1 Referencing Options 179
4.7.2 Distance To Reference Mark 180
4.7.3 Distance Code Inverted 180
4.8 Tuning and Customizing Properties 181
4.8.1 Positioner Movement Type 181
4.8.2 Positioner Is Custom Type 182
4.8.3 Positioner Base Unit 183
4.8.4 Positioner Base Resolution 184
4.8.5 Positioner Sensor Head Type 185
4.8.6 Positioner Reference Type 186
4.8.7 Positioner P Gain 187
4.8.8 Positioner I Gain 188
4.8.9 Positioner D Gain 189
4.8.10 Positioner PID Shift 190
4.8.11 Positioner Anti Windup 191

4.8.12 Positioner ESD Distance Threshold 192
4.8.13 Positioner ESD Counter Threshold 193
4.8.14 Positioner Target Reached Threshold 194
4.8.15 Positioner Target Hold Threshold 195
4.8.16 Save Positioner Type 196
4.8.17 Positioner Write Protection 196
4.9 Streaming Properties 197
4.9.1 Stream Base Rate 197
4.9.2 Stream External Sync Rate 198
4.9.3 Stream Options 199
4.9.4 Stream Load Maximum 200
4.10 Diagnostic Properties 200
4.10.1 Channel Error 200
4.10.2 Channel Temperature 201
4.10.3 Bus Module Temperature 202
4.11 Auxiliary Properties 203
4.11.1 Aux Positioner Type 203
4.11.2 Aux Positioner Type Name 204
4.11.3 Aux Input Select 204
4.11.4 Aux I/O Module Input Index 205
4.11.5 Aux Direction Inversion 207
4.11.6 Aux I/O Module Input0 / Input1 Value 208
4.11.7 Aux Digital Input Value 208
4.11.8 Aux Digital Output Value / Set / Clear 209
4.11.9 Aux Analog Output Value0 / Value1 210
4.12 I/O Module Properties 211
4.12.1 I/O Module Options 211
4.12.2 I/O Module Voltage 213
4.12.3 I/O Module Analog Input Range 213
4.13 Input Trigger Properties 215
4.13.1 Device Input Trigger Mode 215
4.13.2 Device Input Trigger Condition 216
4.14 Output Trigger Properties 217
4.14.1 Channel Output Trigger Mode 217
4.14.2 Channel Output Trigger Polarity 218
4.14.3 Channel Output Trigger Pulse Width 219
4.14.4 Channel Position Compare Start Threshold 220
4.14.5 Channel Position Compare Increment 221
4.14.6 Channel Position Compare Direction 221
4.14.7 Channel Position Compare Limit Min 222
4.14.8 Channel Position Compare Limit Max 223
4.15 Hand Control Module Properties 224
4.15.1 Hand Control Module Lock Options 224
4.15.2 Hand Control Module Default Lock Options 226
4.16 API Properties 227
4.16.1 Event Notification Options 227
4.16.2 Auto Reconnect 228

5 Event Reference 229
5.1 Event Summary 229
5.2 Detailed Event Description 231
5.2.1 None 231
5.2.2 Movement Finished 231
5.2.3 Holding Aborted 231
5.2.4 Sensor State Changed 232
5.2.5 Reference Found 232
5.2.6 Following Error Limit 233
5.2.7 Sensor Module State Changed 233
5.2.8 Over Temperature 233
5.2.9 High Voltage Overload 234
5.2.10 Adjustment Finished 234
5.2.11 Adjustment State Changed 235
5.2.12 Adjustment Update 235
5.2.13 Stream Finished 235
5.2.14 Stream Ready 236
5.2.15 Stream Triggered 236
5.2.16 Command Group Triggered 237
5.2.17 Hand Control Module State Changed 237
5.2.18 Emergency Stop Triggered 238
5.2.19 External Input Triggered 238
5.2.20 Request Ready 238
5.2.21 Connection Lost 239

6 ASCII Interface 241
6.1 Connection Setup 241
6.1.1 Note On Message Termination 242
6.2 SCPI Basics 242
6.2.1 SCPI Conformance Information 242
6.2.2 Command Structure 243
6.2.3 Traversing the Command Tree 244
6.2.4 Queries 245
6.3 Basic Programming Examples 245
6.3.1 Get Property 245
6.3.2 Set Property 246
6.3.3 Calibrate 246
6.3.4 Reference 246
6.3.5 Move 246
6.3.6 Stop 246
6.3.7 Movement State 247
6.3.8 Error Handling 247
6.4 Using Command Groups 249
6.4.1 Command Set 249
6.4.2 Examples 250
6.5 Streaming Trajectories 252
6.5.1 Command Set 252
6.5.2 Example 253

6.6 Command Summary 254
6.6.1 Common Commands 254
6.6.2 Movement Commands 255
6.6.3 Property Command Tree 255
6.7 SCPI Error Codes 260

A Code Definition Reference 262
A.1 Error Codes 262

1 *INTRODUCTION*

This document describes the application programming interface (API) of the SmarAct MCS2. It may be used to control one or more MCS2 devices by software.

While this document mainly serves as a reference when programming your own software it also supplies some background information for a better understanding of the overall system.

1.1 Terminologies

This section defines general terminologies that are used throughout this document. This section only gives a brief summary and the terminologies are explained in more detail later in this document.

Closed-Loop Movements are movements where sensor data is used as feedback to control the position, velocity and/or acceleration of a positioner. To be able to perform closed-loop movements the targeted positioner obviously must be equipped with an integrated position sensor. Furthermore, the sensor must not be disabled. See section 2.6.4 Closed-Loop Movements.

Open-Loop Movements are movements that do not use sensor data as feedback. The positioner simply moves according to the given parameters and the exact distance traveled is undefined. Especially, movements in different directions, but otherwise identical parameters, will typically result in slightly varying traveling distances. See section 2.6.3 Open-Loop Movements.

Calibrating is a process where the controller analyzes the individual characteristics of a positioner in order to optimize closed-loop behavior. The calibration data is saved to non-volatile memory. Therefore, the calibration only needs to be performed when the system setup changes, but not necessarily on each system start-up. See section 2.6.1 Calibrating.

Referencing is a process where the controller moves a positioner to detect its absolute physical position. After the referencing, points of interest identified in previous sessions may easily be recalled. See section 2.6.2 Referencing.

Trajectory Streaming allows to move several positioners synchronously along a defined trajectory. See section 2.15 Trajectory Streaming.

Hold Time The hold time of a closed-loop movement specifies how long the positioner will actively hold its position after reaching the target. This may be useful to compensate drift effects.

Max Closed-Loop Frequency When performing closed-loop movements, the control-loop uses the current position and the commanded target position to generate a driving signal for the piezo actuator taking the control-loop parameters (PID) into account. The maximum allowed frequency that is generated by the control-loop depends on the actual positioner as well as the environment. (E.g. HV and UHV requires lower allowed frequencies.) The max closed-loop frequency defines the upper limit for the generated driving signal.

2 *GENERAL CONCEPTS*

2.1 Connecting and Disconnecting

Before being able to communicate with a device a connection must be established via a call to `SA_CTL_Open`. This function connects to the device specified in the locator parameter (see section 2.1.1) and returns a handle to the device, if the call was successful. The returned device handle must be saved within the application and passed as a parameter to the other API functions. Once the connection is established you can use the other functions to interact with the connected device. If an application requires to connect to more then one device it must open each device separately. The API processes all communication independently for each device handle.

A device that has been acquired by an application cannot be acquired by a second application at the same time. You must close the connection to the device by calling `SA_CTL_Close` before it is free to be used by other applications. Not closing a device will cause a resource leak.

If you have threads blocking on functions like `SA_CTL_WaitForEvent` you may unblock them for a clean shutdown by calling `SA_CTL_Cancel`. The `SA_CTL_WaitForEvent` function will then return with the error code `SA_CTL_ERROR_CANCELED`.

NOTICE
Connecting to a device via the ASCII interface uses a different mechanism. Please refer to section 6.1 for more information.

2.1.1 Locators for Device Identification

Devices are identified with *locator* strings, similar to URLs used to locate web pages. The following sections describe the syntax of these locator strings.

USB Device Locator Syntax

Devices with a USB interface can be addressed with one of the following locator syntaxes:

- `usb:sn:<serial>`
  where `<serial>` is the device serial which is printed on the housing of the device.
  Example: `usb:sn:MCS2-00000412`

- `usb:ix:<n>`
  where the number `<n>` selects the *nth* device in the list of all currently attached devices with a USB interface.
  Example: `usb:ix:0`
  The drawback of identifying a device with this method is that the number and the order of connected devices may change between sessions, so the index `n` may not always refer to the same device. It is only safe to do this if you have exactly one device connected to the PC.

It is recommended to use the first format for USB devices.

Network Device Locator Syntax

Devices with a network interface are addressed with one of the following locator syntaxes:

- `network:sn:<serial>`
  where `<serial>` is the device serial which is printed on the housing of the device.
  Example: `network:sn:MCS2-00000412`
- `network:<ip>`
  where `<ip>` is an IPv4 address which consists of four integer numbers between 0 and 255 separated by a dot.
  Example: `network:192.168.1.200`

NOTICE
Data transmission bandwidth and latencies over networks can vary much more than over e.g. USB. A program should not rely on low transmission latencies.

2.1.2 Finding Devices

Devices may be connected to by using a specific locator as outlined above. To find devices automatically the function `SA_CTL_FindDevices` may be used. It will scan the USB ports as well as the network interfaces and return a list with the locator strings of the found devices.

Note that the Network Discover Mode property (see section 4.2.7) must be configured to active or passive mode to make it possible to list devices with ethernet interface.

2.1.3 Network Interface Configuration

While devices with USB interface do not need any interface configuration, the ethernet interface must be configured with the network parameters: DHCP mode, IP address, subnet mask and gateway IP address. The MCS2 is delivered with a default IP configuration which may be adjusted to match the users network settings.

The following table lists the default configuration:

Parameter	Default Value
DHCP Mode	disabled
IP Address	192.168.1.200
Subnet Mask	255.255.0.0
Gateway IP	192.168.1.1
Pass-Key	smaract

The interface may be configured to use DHCP to obtain an IP address from a DHCP-server or to use a static IP configuration. The configuration may be changed by connecting to the integrated web server, by using the configuration menu of an MCS2 Hand Control Module or by using the SmarActNetConfig tool for the PC.

See the *MCS2 User Manual* document for more details on the configuration.

2.2 Properties

Properties are configuration values that define the behavior of the device. Each property has a data type and an access mode. Some properties may be read and written, while others are read only or (in rare cases) write only. See chapter 4 "Property Reference" for a list of available properties and their descriptions.

Depending on the data type a property has you must use the corresponding function variant to access it. For example, the Number of Channels property is of type I32. Therefore, you must use the `SA_CTL_GetProperty_i32` function to read the property. In contrast the Device Serial Number property is of type string. Therefore, you must use the `SA_CTL_GetProperty_s` function to read the property.

Properties are identified by a *property key* that must be passed to the function call when accessing a property. Properties are categorized into device, module and channel properties. Module and channel properties require an additional *index* parameter to address a specific module or channel. Note that the index parameter is zero-based. In case of device properties the controller is already addressed by the device handle. Therefore, the index parameter is unused and must be set to zero.

Most properties are non-persistent which means that modifications do not outlive a power cycle. At device start-up they have the default value that is specified in the detailed property description. Other properties are kept persistent in the internal non-volatile memory. Therefore, their values are preserved and loaded at device start-up.

2.3 Accessing Properties

Modifying or retrieving property values takes a major role in controlling a device by software. Therefore, the API offers a variety of functions to get and set property values in order to meet all requirements an application might have. A straight forward method, though easy to use, is somewhat inefficient, while more complicated methods may greatly improve efficiency. The application

may decide on a per-call basis which method to use, thus being very flexible depending on the applications context.

The different methods of accessing properties may be categorized by their use case and are described in the following sections. The figures illustrate the sequence of actions for getting two property values. Green boxes indicate non-blocking API calls while red boxes indicate blocking calls. Setting properties is very similar and is not explicitly discussed.

2.3.1 Synchronous Access

This is the easiest method for accessing properties since it consists of one simple function call for getting one property value (e.g. `SA_CTL_GetProperty_i32`). When the function returns the result is available (see figure 2.1).

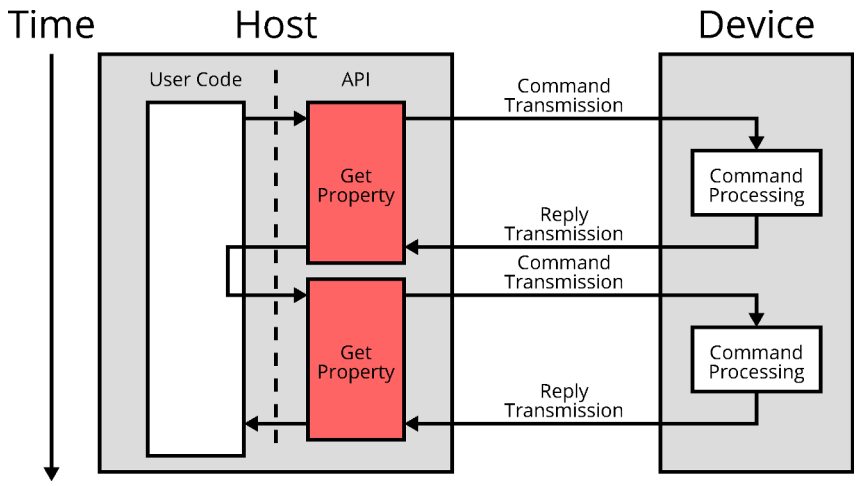


Figure 2.1: Synchronous Property Access

When the API function is called a command is sent to the device and the function waits for a reply from the device before it returns. From the view of the application, the function has a blocking behavior. Depending on the transmission delays the blocking time may be in the range of several milli seconds. During this time the user application cannot perform any other tasks. Therefore, this access method is the slowest of all.

Functions Used

`SA_CTL_GetProperty_i32`, `SA_CTL_SetProperty_i32`

Example Read

```
int32_t value[2];
int8_t channel;
for (channel = 0; channel < 2; channel++) {
  SA_CTL_Result_t result = SA_CTL_GetProperty_i32(
```

```
    dHandle,channel,SA_CTL_PKEY_CHANNEL_STATE,&(value[channel])
  );
  if (result) {
    // handle error
  }
}
// value[0] and value[1] hold the channel state
```

Example Write

```
int32_t value[2] = {SA_CTL_MOVE_MODE_CL_ABSOLUTE,
                    SA_CTL_MOVE_MODE_CL_RELATIVE};
int8_t channel;
for (channel = 0; channel < 2; channel++) {
  SA_CTL_Result_t result = SA_CTL_SetProperty_i32(
    dHandle,channel,SA_CTL_PKEY_MOVE_MODE,value[channel]
  );
  if (result) {
    // handle error
  }
}
```

2.3.2 Asynchronous Access

This method requires two function calls for getting one property value. One for requesting the property value and one for retrieving the answer (see figure 2.2).

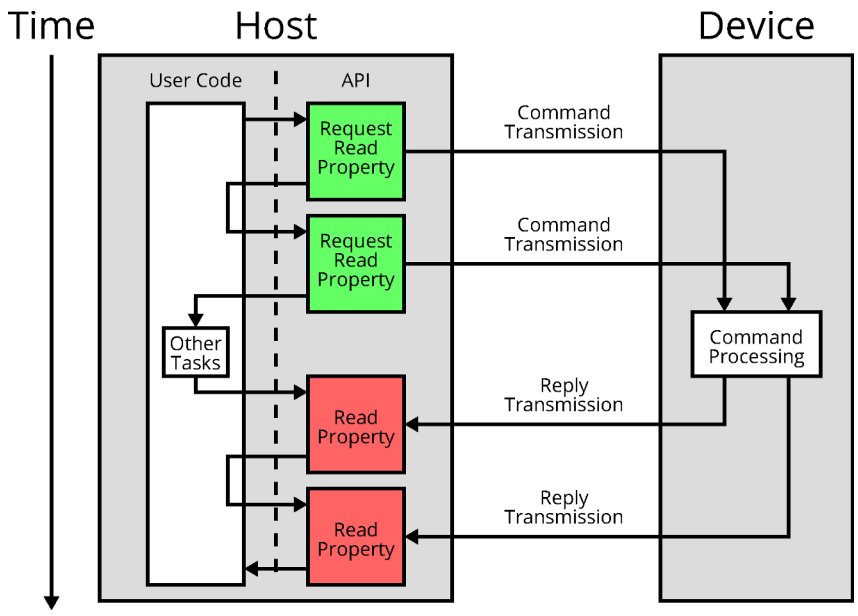


Figure 2.2: Asynchronous Property Access

When the API function is called a command is sent to the device and the function returns immediately, allowing the application to issue another request (or perform other tasks). When the application has finished performing other tasks (or cannot proceed until the property values are available) it may call the API function to receive the result.

The advantage of this method is that the application may request several property values in fast succession and then perform other tasks before blocking on the reception of the results.

Functions Used

`SA_CTL_RequestReadProperty`, `SA_CTL_ReadProperty_i64`, `SA_CTL_RequestWriteProperty_i64`, `SA_CTL_WaitForWrite`

Example Read

```
SA_CTL_Result_t result;
int64_t value[2];             // buffer for values to read
SA_CTL_RequestID_t rID[2];    // buffer for request IDs
int8_t channel;
// issue requests for two channels
  for (channel = 0; channel < 2; channel++) {
  result = SA_CTL_RequestReadProperty(
    dHandle,channel,SA_CTL_PKEY_POSITION,&(rID[channel]),0
  );
  if (result) {
    // handle error
  }
}
// process other tasks
// ...
// retrieve results
for (channel = 0; channel < 2; channel++) {
  result = SA_CTL_ReadProperty_i64(
    dHandle,rID[channel],&(value[channel])
  );
  if (result) {
    // handle error
  }
}
```

Example Write

```
SA_CTL_Result_t result;
SA_CTL_RequestID_t rID[2];    // buffer for request IDs
int8_t channel;
// issue requests for two channels (set position to zero)
```

```
for (channel = 0; channel < 2; channel++) {
  result = SA_CTL_RequestWriteProperty_i64(
    dHandle,channel,SA_CTL_PKEY_POSITION,0,&(rID[channel]),0
  );
  if (result) {
    // handle error
  }
}
// process other tasks
// ...
// retrieve results
for (channel = 0; channel < 2; channel++) {
  result = SA_CTL_WaitForWrite(
    dHandle,rID[channel]
  );
  if (result) {
    // handle error
  }
}
```

2.3.3 High-Throughput Asynchronous Access

This method is similar to the asynchronous access with the difference that request commands are bundled (see figure 2.3).

When the API function is called the request is buffered. The function returns immediately and the command transmission is held back until the buffer is flushed. Again, the application may request several property values in fast succession and then perform other tasks before blocking on the reception of the results. In addition, the underlying media is able to combine several requests into one packet, thus further optimizing communication delays.

Functions Used

`SA_CTL_CreateOutputBuffer`, `SA_CTL_FlushOutputBuffer`, `SA_CTL_RequestReadProperty`, `SA_CTL_ReadProperty_i64`

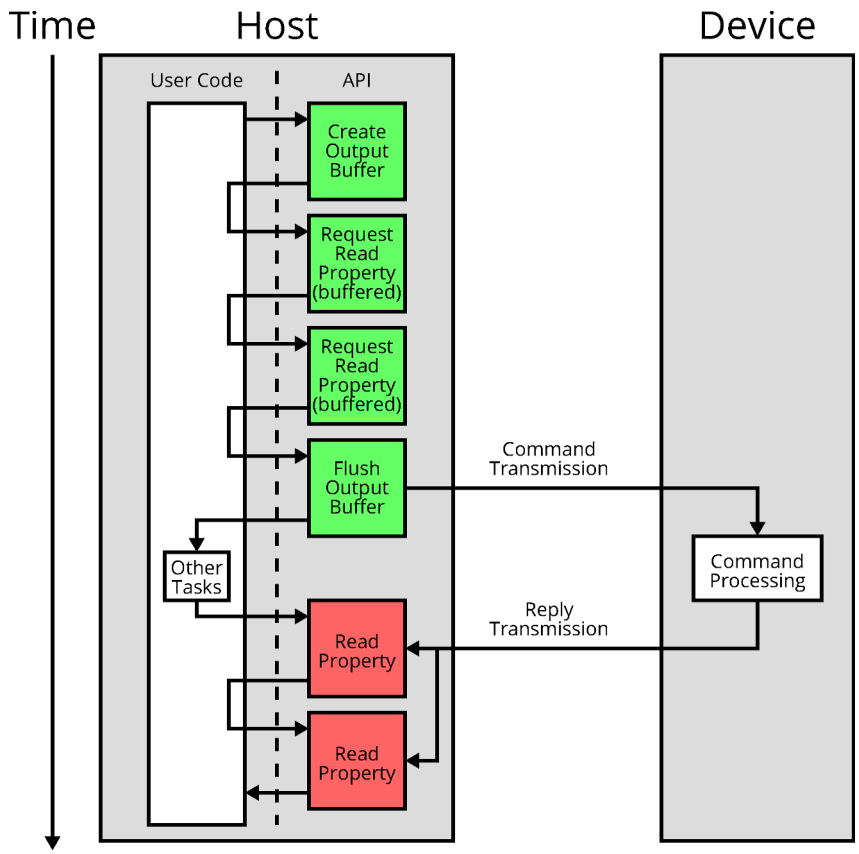


Figure 2.3: High-Throughput Asynchronous Property Access

Example Read

```
SA_CTL_Result_t result;
int32_t value[2];              // buffer for values to read
SA_CTL_RequestID_t rID[2];     // buffer for request IDs
int8_t channel;
// create output buffer
SA_CTL_TransmitHandle_t tHandle;
result = SA_CTL_CreateOutputBuffer(dHandle,&tHandle);
if (result) {
  // handle error
}
// issue requests for two channels
for (channel = 0; channel < 2; channel++) {
  // by passing the transmit handle (instead of zero)
  // the request is associated with the output buffer and
  // therefore only sent when the buffer is flushed (see below)
  result = SA_CTL_RequestReadProperty(
    dHandle,channel,SA_CTL_PKEY_POSITION,&(rID[channel]),tHandle
  );
  if (result) {
    // handle error
```

```
  }
}
// flush output buffer
SA_CTL_FlushOutputBuffer(dHandle, tHandle);
// process other tasks
// ...
// retrieve results
for (channel = 0;channel < 2; channel++) {
  result = SA_CTL_ReadProperty_i64(
    dHandle,rID[channel],&(value[channel])
  );
  if (result) {
    // handle error
  }
}
```

2.3.4 Call-and-Forget Mechanism

For property writes the result is only used to report errors. With the call-and-forget mechanism the device does not generate a result for writes and the application can continue processing other tasks immediately. Compared to asynchronous accesses, the application doesn't need to keep track of open requests and collect the results at some point. This mode should be used with care so that written values are within the valid range.

The call-and-forget mechanism is used by passing a null pointer for the request ID pointer to the `SA_CTL_RequestWriteProperty_x` functions.

Functions Used

`SA_CTL_RequestWriteProperty_i64`

Example Write

```
SA_CTL_Result_t result;
int8_t channel;
// issue requests for two channels (set position to zero)
for (channel = 0; channel < 2; channel++) {
  result = SA_CTL_RequestWriteProperty_i64(
    dHandle,channel,SA_CTL_PKEY_POSITION,0,NULL,0
  );
  if (result) {
    // handle error
  }
}
```

2.3.5 Request Ready Notification

Instead of using the blocking `SA_CTL_ReadProperty_x`/`SA_CTL_WaitForWrite` functions to retrieve the result of an asynchronous request, the event system (see section 2.4 "Event Notifications") can be used to get a notification once the answer has been received from the device. After receiving a Request Ready event (see there) the result of the asynchronous operation can be retrieved without blocking using the functions mentioned above.

Note that the request ready event needs to be enabled using the Event Notification Options property.

Example Request

```
// enable request ready events
result = SA_CTL_SetProperty_i32(
  dHandle, 0, SA_CTL_PKEY_EVENT_NOTIFICATION_OPTIONS,
  SA_CTL_EVT_OPT_BIT_REQUEST_READY_ENABLED
);
if (result) { /* handle error */ }

// send asynchronous request
SA_CTL_RequestID_t rID;
result = SA_CTL_RequestReadProperty(
  dHandle, 0, SA_CTL_PKEY_CHANNEL_STATE, &rID, 0
);
if (result) { /* handle error */ }
```

Example Event Processing

```
SA_CTL_Event_t evnt;
result = SA_CTL_WaitForEvent(dHandle, &evnt, SA_CTL_INFINITE);
if (result) { /* handle error */ }

if (evnt.type == SA_CTL_EVENT_REQUEST_READY) {
  // extract event data
  SA_CTL_RequestID_t rID = SA_CTL_EVENT_REQ_READY_ID(evnt.i64);
  int requestType = SA_CTL_EVENT_REQ_READY_TYPE(evnt.i64);
  int dataType = SA_CTL_EVENT_REQ_READY_DATA_TYPE(evnt.i64);

  // process read results
  if (requestType == SA_CTL_EVENT_REQ_READY_TYPE_READ) {
    size_t arraySize = SA_CTL_EVENT_REQ_READY_ARRAY_SIZE(evnt.i64);
    switch (dataType) {
      case SA_CTL_DTYPE_INT32:
      {
        std::vector<int32_t> values(arraySize);
        result = SA_CTL_ReadProperty_i32(
```

```
            dHandle, rID, values.data(), &arraySize
          );
          if (result) { /* handle error */ }

          values.resize(arraySize);
          // property data is now stored in values
          break;
        }
        // handle other data types
      }
    }
  }
```

2.4 Event Notifications

In some situations events might occur that require further attention or reactions by the user. To avoid that the application has to poll the occurrence of such events the MCS2 offers a notification system. If an event occurs the MCS2 generates a notification event informing about the situation.

The application may receive events using the `SA_CTL_WaitForEvent` function. It returns events in form of a pointer to the struct:

```
typedef struct {
    uint32_t idx;
    uint32_t type;
    union {
        int32_t i32;
        int64_t i64;
        uint8_t unused[24];
    };
} SA_CTL_Event_t;
```

The fields of the struct have the following meaning:

- `idx` holds the source index that the event originated from. This may be a device, module or channel index, depending on the event type.
- `type` holds the type of the event. See chapter 5 "Event Reference" for a detailed description of the events and their parameters.
- `i32` / `i64` / `unused` are parameter fields that further describe the event. The meaning depends on the event type.

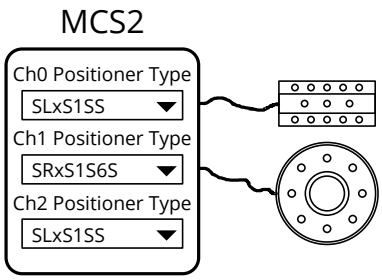
While the event type indicates "what happened" the event parameter gives a more detailed hint why the event occurred. An event can also be translated into a human readable string by using the `SA_CTL_GetEventInfo` function.

Note that all **device events** are enabled by default. There is no property to explicitly enable or disable any specific device events. Only **API events** are disabled by default and need to be enabled explicitly by configuring the Event Notification Options property.

The `SA_CTL_Cancel` function can be used to abort a waiting `SA_CTL_WaitForEvent` call.

2.5 Positioner Types

During the configuration of the device each channel must be configured with the type of positioner that is connected to the channel. The positioner type implicitly gives the controller information about how to calculate positions, handle the referencing, configure the control-loop, etc.



The MCS2 controller provides sets of standard configuration parameters for all kinds of SmarAct positioners. For the majority of applications these predefined types are sufficient. To configure a positioner type to a channel simply set the Positioner Type property.

NOTICE
When the positioner type of a channel is changed, the channel must be calibrated to ensure proper operation of the positioner. See section 2.6.1 "Calibrating" for more information.

Each channel stores the positioner type setting to non-volatile memory. Consequently, there is no need to configure the positioner type for each session. Only when changing the physical setup (switching positioners etc.) you must reconfigure (and calibrate) the channel again. Note that the positioner type is represented by a generic *type code* instead of the descriptive name string. This type code must be written to the Positioner Type property to configure the type. The descriptive name may be read with the Positioner Type Name property. Furthermore, the Tuning and Customizing Properties may be used to read additional information of the configured positioner type.

Please refer to the *MCS2 Positioner Types* document for a list of available positioner types.

2.5.1 Custom Positioner Types

In special cases it might be necessary to modify tuning parameters of a positioner type to adapt to an application perfectly. The MCS2 controller offers this possibility by giving access to the tuning parameters. Once the tuning is finished the set of parameters may be saved to a *custom positioner type* slot. As a safety feature, all tuning properties are write protected by default. This prevents accidental modification of any parameters. A special key must be written to the Positioner Write

Protection property to unlock the write access to the tuning properties. As long as the write protection is active, writing to a tuning property will return a `SA_CTL_ERROR_PERMISSION_DENIED` error.

Custom positioner type slots are also used to define the control-loop parameters in case an auxiliary input is used as feedback signal for the control-loop. Refer to section 2.16.5 "Using Analog Inputs as Control-Loop Feedback" for more information.

Creating Custom Positioner Types

When tuning a positioner type the first step should be to select one of the predefined positioner types to act as a template. Note that this step is important to define several internal parameters which are not user accessible. The predefined positioner type defines e.g. the sensor type (S, L, M, etc.) and sensor supply voltage as well as the position calculation parameters. After this, tuning parameters may be modified. As long as the modified positioner type was not saved to a custom slot, the positioner type is read as 0 to indicate that the modifications are volatile. (The Positioner Type Name property returns 'modified' in that case.) Powering down the device in this state will discard the changes made. To save the modified set of parameters use the Save Positioner Type property. This will save the settings to one of four custom positioner type slots and set the Positioner Type to the new custom type implicitly.

CAUTION
Configuring inappropriate values may result in unstable or unexpected behavior of the positioners and potential damage of the stage. Custom tuning must be used with caution!

The available properties for customizing a positioner type are described in section 4.8 "Tuning and Customizing Properties".

2.6 Moving Positioners

There are several commands available that induce a movement of a positioner (**movement commands**). Mainly these are:

- Calibrating (`SA_CTL_Calibrate`).
- Referencing (`SA_CTL_Reference`).
- Moving (`SA_CTL_Move`). Depending on the configured Move Mode this command covers:
  - Open-loop movements (Scanning and Stepping)
  - Closed-loop movements
- Stopping (`SA_CTL_Stop`).

These commands are described in the following sections.

Generally, the base unit for position values is pico meters (pm) for linear positioners and nano degrees (n°) for rotary positioners.

NOTICE

API functions that involve movement of positioners (such as `SA_CTL_Move`, `SA_CTL_Calibrate` and `SA_CTL_Reference`) are always sent to the device asynchronously. Therefore, these functions do not return an acknowledge or error directly. Instead, the movement commands will always generate a `SA_CTL_EVENT_MOVEMENT_FINISHED` event where the event parameter indicates success or failure. For example, if a closed-loop movement could not be started due to a missing sensor, the event parameter will be `SA_CTL_ERROR_NO_SENSOR_PRESENT`. See section 2.6.7 "Movement Feedback" for more information.

2.6.1 Calibrating

Even though every positioner is categorized by its type (which is configured to the channel via the Positioner Type property, see also section 2.5 "Positioner Types") each individual positioner may have slightly different characteristics that require the tuning of some internal parameters for correct operation and optimal results.

The `SA_CTL_Calibrate` function is used to adapt to these characteristics and automatically detects parameters for an individual positioner. It must be called once for each channel if the mechanical setup changes (different positioners connected to different channels). The calibration data will be saved to non-volatile memory. If the mechanical setup is unchanged, it is not necessary to run the calibration on each initialization, but newly connected positioners have to be calibrated in order to ensure proper operation.

The calibration routine is only executable by a positioner that has a sensor attached to it. The sensor must also be enabled or in power save mode (see the Sensor Power Mode property). Otherwise the `SA_CTL_EVENT_MOVEMENT_FINISHED` event that is generated by the channel will hold an error code as parameter. The calibration takes a few seconds to complete. During this time the Channel State bit `SA_CTL_CH_STATE_BIT_CALIBRATING` is set.

CAUTION

As a safety precaution, make sure that the positioner has enough freedom to move without damaging other equipment.

Before calling the `SA_CTL_Calibrate` function the Calibration Options property should be configured to define the behavior of the calibration sequence. This property holds a bit mask which is outlined in the following table.

Bit	Name	Short Description
0	Direction	Defines the direction in which the positioner will move for calibration purposes.
1	Detect Distance Code Inversion	Activates a special mode that detects the individual setup of positioners with multiple reference marks. For normal calibration this bit should be set to 0.
2	Advanced Sensor Correction	Activates a calibration routine to compensate periodic sensor errors.
8	Limited Travel Range	Allows more than one endstop while calibrating. Should be used for positioners with limited travel range, e.g. micro grippers.
3 .. 7, 9 .. 31	Reserved	These bits are reserved for future use.

Signal Correction Calibration (calibration options `0x00` or `0x01`)

During this calibration routine the positioner will perform a movement of up to several mm in the configured direction to optimize the position calculation for the sensor signals of the positioner. The signal correction calibration should not be started near a mechanical end stop. Nonetheless the calibration sequence automatically detects an endstop and reverts the movement direction to continue the calibration in the opposite direction. If more than one endstop is detected the calibration sequence is aborted with an error.

Some positioners (e.g. micro grippers) have a very **limited travel range**. For these positioners the movement distance may be too small to successfully finish the calibration.

The `SA_CTL_CALIB_OPT_BIT_LIMITED_TRAVEL_RANGE` calibration options flag may be used to increase the number of allowed endstops while calibrating. The calibration sequence then moves back and forth between the two endstops to perform the signal corrections.

Positioners that are referenced via a **mechanical end stop** (see section 2.7.4 "Positioners With Endstop Reference") are moved to the end stop as part of the calibration routine. For this movement the configured Move Velocity and Move Acceleration are used.

Which end stop is used for referencing is defined by the configured Safe Direction instead of the direction bit of the Calibration Options property. Note that when changing the Safe Direction the end stop must be calibrated again for proper operation.

As a safety precaution, make sure that the positioner has enough freedom to move without damaging other equipment.

Distance Code Inversion Detection (calibration options `0x02` or `0x03`)

This calibration routine may be used to correct the absolute position calculation when referencing positioners with multiple reference marks. In rare cases the reference algorithm may produce faulty results due to a reference coding mismatch. These situations may be resolved by executing this calibration routine.

Advanced Sensor Correction Calibration (calibration options `0x04` or `0x05`)

This calibration routine is used to improve the absolute sensor accuracy by compensating the periodic sensor error. A calibration sequence is needed to generate a compensation table which is stored in the controller. This calibration must be performed for every channel that should use the advanced sensor correction. During this calibration routine the positioner will perform a movement of up to several mm in the configured direction. The compensation may then be activated by setting the `SA_CTL_SIGNAL_CORR_OPT_BIT_ASC` bit of the Signal Correction Options property.

NOTICE
The advanced sensor correction needs a feature permission to be activated on the controller. See section 2.19 "Feature Permissions" for more information.

2.6.2 Referencing

The `SA_CTL_Reference` function may be used to instruct a positioner to determine its physical position. It will start to move in the configured search direction and look for a reference. The positioner must have a sensor attached to it and the sensor must be enabled or in power save mode in order to perform the referencing sequence (see the Sensor Power Mode property).

Depending on the reference strategy (which is partly predefined by the positioner type and partly configurable) as well as the individual positioner, the referencing takes some time to complete. During this time the Channel State bit `SA_CTL_CH_STATE_BIT_REFERENCING` is set. In case the reference could not be found the `SA_CTL_EVENT_MOVEMENT_FINISHED` event that is generated by the channel will hold an error code as parameter.

Before calling the `SA_CTL_Reference` function the Referencing Options property can be configured to define the behavior of the reference sequence. This property holds a bit mask with several options that influence the strategy of how to find the reference. Please refer to section 2.7.1 "Reference Marks" for more information.

Note that reference movements (when successful) generate two events. One when the reference position has been determined and one after the positioner has come to a stop. The first event is mainly useful when using the *Continue On Reference Found* feature (see section 2.7.1 "Reference Marks").

2.6.3 Open-Loop Movements

There are two types of open-loop movement:

- *Scan movements* allow to control the deflection of the piezo element of the positioner directly. To perform scan movements the Move Mode property must be set to one of the values `SA_CTL_MOVE_MODE_SCAN_ABSOLUTE` or `SA_CTL_MOVE_MODE_SCAN_RELATIVE`.

  The scan velocity may be specified with the Scan Velocity property. The `SA_CTL_Move` function must be called to start the actual scan movement. The *move value* parameter of the

SA_CTL_Move function is then interpreted as target scan position to which to scan to, respectively scan target increment in case of relative scan movement. The valid range for the scan position is 0 ... 65 535 for absolute scan positions and −65 535 ... 65 535 for relative scan increments. Note that for relative scan movements the movement will stop at the boundary if the resulting absolute scan target exceeds the valid range.

- *Step movements* allow to perform a burst of steps with the given frequency and amplitude. To perform step movements the Move Mode must be set to SA_CTL_MOVE_MODE_STEP. Frequency and amplitude of the generated output signal may be specified with the properties Step Frequency and Step Amplitude. The SA_CTL_Move function must be called to start the actual step movement. The *move value* parameter of the SA_CTL_Move function is then interpreted as number of steps. The sign of the value codes the movement direction. The valid range for the step parameter is −100 000 ... 100 000.

The Channel State bit SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING is set while performing scan or step movements.

2.6.4 Closed-Loop Movements

In order to perform a closed-loop movement the positioner must have a sensor attached to it. The sensor must also be enabled or in power save mode (see the Sensor Power Mode property). If this is not the case the SA_CTL_EVENT_MOVEMENT_FINISHED event that is generated by the channel will hold an error code as parameter.

Before calling the SA_CTL_Move function the Move Mode property must be set to one of the following values:

- SA_CTL_MOVE_MODE_CL_ABSOLUTE In this mode the *move value* that is passed to the SA_CTL_Move function is interpreted as the new absolute target position the positioner should move to.
- SA_CTL_MOVE_MODE_CL_RELATIVE In this mode the *move value* that is passed to the SA_CTL_Move function is added to the current (target) position. The *move value* of 0 has a special meaning in this mode: the channel aborts an ongoing movement and actively holds the current position.

Additionally, the following properties may be configured to modify the behavior of the closed-loop movement (see also the detailed property descriptions in chapter 4):

- Move Velocity and Move Acceleration
  These properties define the velocity resp. the acceleration with which the closed-loop movement is performed. If the move velocity is set to zero (default) then the velocity control is disabled and the positioner moves to the target position as fast as possible, more precisely, only limited by the *maximum closed-loop frequency* (see Max Closed Loop Frequency). Likewise, if the acceleration is set to zero (default) then the acceleration control is disabled and the positioner accelerates and decelerates as fast as possible (only limited by mechanical factors).

- Max Closed Loop Frequency
  Generally, the channel will not drive the positioner with frequencies above the maximum allowed frequency. If the maximum frequency is set too low for a certain move velocity, then the move velocity might not be reached or held. In this case the maximum frequency must be increased. Be aware that different positioners reach different velocities. If a positioner is not able to move as fast as the configured move velocity, then the driver will cap at the maximum driving frequency.

- Hold Time
  The channel may be instructed to hold the target position after it has been reached. This may be useful to compensate for drift effects and the like. The positioner will implicitly adjust the deflection of the piezo to hold the position if needed. When the piezo element of the positioner reaches a boundary a single step is performed. While holding the position the Channel State bit `SA_CTL_CH_STATE_BIT_CLOSED_LOOP_ACTIVE` is set and the bit `SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING` is cleared. After the hold time elapsed the channel is stopped.

  Note that the closed-loop movement is considered finished as soon as the target position is reached and *not* when the optional hold time has elapsed.

  The endstop detection is still active in holding state. If a positioner is moved away from the target position by external forces and the channel is not able to hold the target position for a longer time an endstop is triggered. A `SA_CTL_EVENT_HOLDING_ABORTED` event is generated to notify about this and the channel is stopped.

- Control Loop Input
  This property defines the feedback signal for the control-loop.

  - `SA_CTL_CONTROL_LOOP_INPUT_DISABLED` The closed-loop operation is disabled. A `SA_CTL_ERROR_CONTROL_LOOP_INPUT_DISABLED` error will be generated when trying to command a closed-loop movement.
  - `SA_CTL_CONTROL_LOOP_INPUT_SENSOR` The channel uses the integrated sensor of a positioner to calculate the current position. This position is used as input signal for the control-loop to allow closed-loop position control.
  - `SA_CTL_CONTROL_LOOP_INPUT_AUX_IN` The input signal of an auxiliary input (e.g. an analog input of an MCS2 IO module) is used as control-loop input.

- Actuator Mode
  This mode defines the type of actuator driving signal generation.

  - `SA_CTL_ACTUATOR_MODE_NORMAL` The normal mode is the default mode. It offers open-loop step movement as well as closed-loop movement.
  - `SA_CTL_ACTUATOR_MODE_QUIET` The quiet mode only allows to perform closed-loop movement and reduces the noise that is emitted from the positioners while moving. It is useful in applications where the noise emission is disturbing. The trade-off between the quiet and the normal mode is the higher heat-dissipation of the controller in quiet mode. For this reason the quiet mode is not recommended for continuous operation.

- `SA_CTL_ACTUATOR_MODE_LOW_VIBRATION` The low vibration mode allows to perform closed-loop movements which produce as little vibrations as possible. It is useful for applications where the high-frequent vibrations of the stick-slip driving principle cause troubles.

NOTICE
The low vibration mode needs a feature permission to be activated on the controller. See section 2.19 "Feature Permissions" for more information.

- Positioner Control Options
  This property defines several options that apply to closed-loop movements. The property value is a bit field containing the following independent flags:

Bit	Name	Short Description
0	Accumulate Relative Position Disabled	Disables the relative position accumulation.
1	No Slip	Forbid the execution of actuator slips (steps).
2	No Slip While Holding	Forbid the execution of actuator slips (steps) only while holding the target position.
3	Forced Slip Disabled	Disables the forced slip feature.
4	Stop On Following Error	Stop positioner if a following error was detected.
5	Target To Zero Voltage	The driver output voltage is forced to zero while retaining the target position after a closed-loop movement.

Undefined flags are unused but might get a meaning in future updates. Undefined flags should be set to zero. The flags have the following meaning:

Accumulate Relative Positions Disabled (Bit 0) This flag affects the behavior of a positioner if a relative position command is issued before a previous one has finished. If relative position commands are to be accumulated (bit cleared, default) then all new relative position commands are added to the previous target position. Otherwise (bit set) the movement is executed relative to the position of the positioner at the time of command arrival.

Example: Say the positioner is currently at its zero position. Two relative movement commands are issued in fast succession both with +1 mm as relative target. With accumulation enabled (default) the final position will be 2 mm. With accumulation disabled the final position will vary (e.g. 1.12 mm) depending on when the second command arrives at the controller.

No Slip (Bit 1) If this flag is set the actuator driving signal generation will never generate slips (steps). This means that only scan movement in the range of the piezo is performed for targeting. It might be useful for applications where the vibration of the piezo slip is unwanted, e.g. while approaching to a probe in the sub micrometer range.

No Slip While Holding (Bit 2) This flag affects the behavior of a positioner if it is instructed to hold the target position after reaching it (see the Hold Time property). The piezo deflection will be adjusted automatically to hold the position. Additionally it may become necessary to do further steps to hold the position if the deflection of the piezo reaches a boundary. However, if this is not desired, this flag may be used to forbid the execution of steps even if this means that the position can not be held. Note that this flag has no effect if the **No Slip** flag (Bit 1) or the **Target To Zero Voltage** flag (Bit 5) is active.

Forced Slip Disabled (Bit 3) When reaching a target position the channel will try to stop at approx. 50% of its step size, thus improving the holding feature. This is achieved by forcing a slip, just before reaching the target position. If this behavior is unwanted it can be disabled with this flag.

Stop On Following Error (Bit 4) This flag defines if a closed-loop movement should be stopped as soon as the configured following error is exceeded. Note that this flag has no effect for movements without velocity control or if the Following Error Limit is set to zero.

Target To Zero Voltage (Bit 5) If this flag is set a special holding sequence is started after a target position was reached. The controller will then perform several piezo scan operations to force the output voltage to zero while retaining the target position. This feature is e.g. useful for applications where the positioner should be moved to a specific target position and then should be disconnected from the controller without additional movement of the positioner carriage. (Which usually happens due to the contraction of the piezo element while discharging from the holding voltage.) Note that the hold threshold for this feature may be configured with the Target To Zero Voltage Hold Threshold property. If a Hold Time is specified the sequence is repeated whenever the difference between current position and target position exceeds the configured hold threshold.

Once configured, call the `SA_CTL_Move` function to start the actual movement. While executing a closed-loop movement the Channel State bits `SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING` and `SA_CTL_CH_STATE_BIT_CLOSED_LOOP_ACTIVE` are set.

2.6.5 Stopping Movements

The `SA_CTL_Stop` function may be used to stop any ongoing movement. It also stops the hold position feature of a closed-loop command. Note that for closed-loop movements with enabled acceleration control a "stop" command instructs the positioner to come to a halt by decelerating to zero velocity. A second "stop" command triggers a hard stop.

To command the channel to abort an ongoing movement and actively hold the current position ("enter holding"), set the Move Mode property to `SA_CTL_MOVE_MODE_CL_RELATIVE` and issue a `SA_CTL_Move` command with its move value parameter set to zero. The Hold Time property must be set to a non-zero value, otherwise the channel is stopped immediately without actively holding the position.

A digital input of an I/O module may be used to issue an emergency stop of all channels. See section 2.17.2 "Emergency Stop Mode" for more information.

2.6.6 Overwriting Movement Commands

Generally, the function calls for movement commands return as soon as the command has been transmitted to the hardware; the calls do not block as long as the command is in execution. Therefore, the software is free to issue new commands to the hardware (potentially to other channels) while the movement is being performed. In particular, new movement commands may also be sent to the same channel at any time. This will cause the previous movement command to be implicitly aborted. Note that there is no need to explicitly stop a channel before sending a new movement command. The new command will simply overwrite the current one.

Note on working with events: Overwriting movement commands (sending movement commands before the command finished event of the previous command has arrived) leads to a race condition. The second command might arrive just before the first has completed, thus, only one command complete event is generated (when the second command completes). However, if the second command arrives just after the first has completed, two command complete events are generated (one for each command).

Note on working with a Hand Control Module: Special care must be taken when using a hand control module and a software running on a PC at the same time. The hand control module sets several movement relevant properties (like move velocity, move acceleration, hold time, step frequency, step amplitude, etc.) prior to commanding a movement command. Thus user software must not rely on previously configured parameters since they may have been modified in the meantime by the hand control module. To be on the safe side, user software may set the Hand Control Module Lock Options property to disable the control inputs of the hand control module while its operation.

2.6.7 Movement Feedback

Movement commands are generally executed asynchronously by the device. Particularly, the API functions do not block for the duration of the execution of the movement. Instead, the functions simply trigger the start of the movement and the software may perform other tasks while the positioner is in motion (e.g. tracking the movement and continuously display the current position).

When issuing movement commands it is usually desirable to know if the movement could successfully be started and especially when the controller has finished the movement (e.g. found the reference mark, reached the target position, etc.). Generally, there are two methods of acquiring this information:

- Polling the Channel State property
- Listening to events

Polling

The Channel State property always indicates the current state of the channel. It may be used to check whether the positioner is moving, holding, stopped etc. The four lower state bits are of interest in this context. The following table summarizes the valid combinations and their meanings:

Bit 3 Referencing	Bit 2 Calibrating	Bit 1 Closed Loop Active	Bit 0 Actively Moving	Activity
0	0	0	0	Stopped
0	0	0	1	Performing an open-loop movement (stepping or scanning)
0	0	1	0	Holding the current target position (after a closed-loop movement)
0	0	1	1	Performing a closed-loop movement (moving to target position)
0	1	0	1	Performing a calibration sequence
1	0	1	1	Performing a reference sequence

Since movement commands are always sent asynchronously to the device, they do not return an acknowledge or error directly. Instead, events are generated. (See next section.)

If event notifications are not used, the success or failure of a movement command may be determined by monitoring the `SA_CTL_CH_STATE_BIT_MOVEMENT_FAILED` bit of the Channel State property. The flag is set to zero if the movement could successfully be started. If the flag is read as one an error occurred. The movement could not be started or the execution failed. The reason for the failure may then be determined by reading the Channel Error property. Note, that the channel error is reset to `SA_CTL_ERROR_NONE` by reading the property.

Further state flags may be monitored to indicate if the execution of a movement could not finish. (E.g. if an endstop was detected while executing the movement). Their meaning is described in section 2.8.3 "Channel State Flags".

Events

Generally, every movement command (including calibrating and referencing) generates an event of type `SA_CTL_EVENT_MOVEMENT_FINISHED` when the execution has finished. Note that a movement is also considered as "finished" if it could not be started due to an error, e.g. an invalid parameter or a closed-loop movement could not be executed, because the sensor is offline. In any case the event parameter will indicate the result of the movement execution. The following event parameters are possible:

Parameter	Meaning
SA_CTL_ERROR_NONE	The movement finished with no error. In this case the event occurs at the time when the movement has finished, e.g. when reaching the target position.
SA_CTL_ERROR_INVALID_PARAMETER	The movement could not be executed because a parameter was invalid.
SA_CTL_ERROR_ABORTED	The movement was started, but then aborted by a stop command. In this case the event occurs at the time the controller received the stop command.
SA_CTL_ERROR_NO_SENSOR_PRESENT, SA_CTL_ERROR_SENSOR_DISABLED	The closed-loop movement could not be started, because no sensor is (currently) available.
SA_CTL_ERROR_POWER_SUPPLY_DISABLED, SA_CTL_ERROR_AMPLIFIER_DISABLED	The movement could not be started, because the power supply / amplifier is disabled.
SA_CTL_ERROR_END_STOP_REACHED	The closed-loop movement was started, but could not be finished normally, because an end stop was encountered.
SA_CTL_ERROR_FOLLOWING_ERR_LIMIT	The closed-loop movement was started, but could not be finished normally, because an following error limit was exceeded.
SA_CTL_ERROR_RANGE_LIMIT_REACHED	The closed-loop movement was started, but could not be finished normally, because a range limit was reached.
SA_CTL_ERROR_BUSY_STREAMING	The movement could not be started, because the channel is currently participating in a trajectory stream.

The full list of error codes may be found in the appendix A.1 "Error Codes".

2.7 Defining Positions

Since position calculation is done on an incremental basis, the MCS2 controller has no way of knowing the physical position of a positioner after a system power-up. It simply assumes its starting position as the zero position.

However, in many applications it is convenient to define a certain physical position as the zero position. The Position property may be set for this purpose. It defines the current position to have an arbitrary value. This can be the zero position or any other position (it is possible to have the zero position outside the complete travel range of the positioner).

Figure 2.4 shows an example of a linear positioner. (a) shows the situation after a system power-up. The positioner assumes its current position as zero. (b) shows the situation after the Position property was set. The current position has been defined to +3 mm and the measuring scale is shifted accordingly.

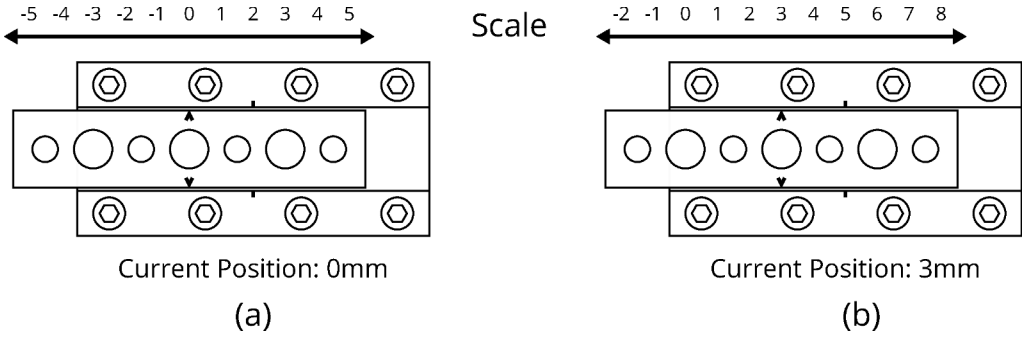


Figure 2.4: Scale Shift

2.7.1 Reference Marks

In the example above the physical position of a positioner must be determined by some external method and then configured to the system. Moreover, this procedure must be done on every system power-up.

To overcome this inconvenience the `SA_CTL_Reference` function may be used to determine the physical position in an automated fashion. After this the controller will return position values according to the positioner's physical measuring scale (but see section 2.7.5 "Shifting the Measuring Scale").

Regarding the referencing, positioner types fall into one of three possible categories:

- **Single Reference Marks** The reference mark of positioners with a single mark is usually located near the middle of the travel range. The positioner will have to move to this mark in order to know its physical position.
- **Multiple Reference Marks** Positioners of this type may calculate their physical position by measuring the distance between two adjacent marks. This has the advantage that the positioner typically only has to move a few milli meters before knowing its physical position which is exceptionally useful when using positioners with very long travel ranges.
- **Endstop Reference Type** Positioners without any reference marks may use the mechanical endstop at the end of their travel range as a known physical position.

The behavior of the positioner while referencing depends on the *positioner type* that is attached to the channel (see Positioner Type property) as well as the configured *referencing options* (see Referencing Options property). The referencing options modify the behavior of the referencing algorithm. Currently, the following bits are available:

Table 2.1 – Referencing Options Bits

Bit	Name	Short Description
0	Start Direction	Defines the direction in which the positioner will start to look for a reference.

Continued on next page

Table 2.1 – *Continued from previous page*

Bit	Name	Short Description
1	Reverse Direction	Only relevant for positioners that have multiple reference marks. Will reverse the search direction as soon as the first reference mark is found.
2	Auto Zero	The current position is set to zero upon finding the reference position.
3	Abort On End Stop	Will abort the referencing on the first end stop that is encountered.
4	Continue On Reference Found	Will not stop the movement of the positioner once the reference is found. The positioner must be stopped manually.
5	Stop On Reference Found	Will stop the movement of the positioner immediately after finding the reference.
6 .. 31	Reserved	These bits are reserved for future use.

NOTICE

Basically, the different mode flags may be combined to obtain a flexible behavior when referencing positioners. However, bits 4 and 5 cannot be combined. If both bits are set then the Stop On Reference Found (bit 5) has priority over Continue On Reference Found (bit 4). See the detailed description of the mode flags below.

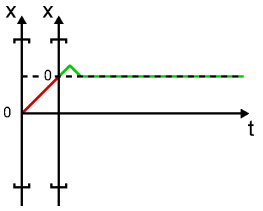
When the `SA_CTL_Reference` command has completed successfully, the system knows the physical position of the positioner (see `SA_CTL_CH_STATE_BIT_IS_REFERENCED` of the Channel State property).

2.7.2 Positioners With Single Reference Marks

This section describes the behavior while referencing positioners with only one reference mark in more detail. The images on the right side illustrate the behavior of an example positioner that is being referenced. The vertical x-axis represents the travel range of the positioner. The square brackets indicate mechanical end stops. The dashed line indicates the position of the reference mark.

In the examples the positioner always starts at position 0 and the physical position is unknown (red line). Once the reference mark has been found the physical position will become known (green line). It is assumed that the physical zero position is on the reference mark.

Default Behavior (reference mode `0b00000000`)

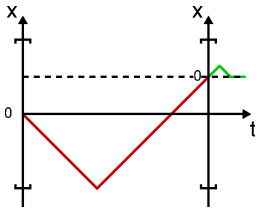


By default the positioner will start to move in forward (positive) direction and look for a reference mark. As soon as the positioner has passed over the reference mark the internal position will be updated. This is indicated by the second x-axis having a different scale shift.

The small overshoot represents the reaction time of the positioner stopping. The amount of the overshoot depends on factors like the velocity with which the referencing is performed, the mass that is mounted on the positioner or a possibly configured acceleration control (in which case it takes some time to decelerate the positioner).

The positioner will turn around and move to the exact location of the reference mark. After this the referencing is complete.

Inverted Start Direction (reference mode `0b00000001`)

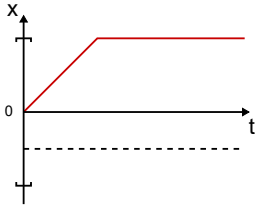


Same as the default referencing, with the difference that the positioner will start to move in backward direction and look for a reference mark.

In this example the positioner will encounter a physical end stop before finding the mark. The positioner will automatically reverse its search direction at the end stop and continue to look for the reference mark.

Note: If the positioner encounters a second end stop then the reference algorithm will be aborted. The positioner is stopped and an error event is generated. Reasons for this situation may be a mechanical or electrical defect (the controller does not register the reference signal for some reason) or the reference mark is outside the physical range of the positioner (e.g. the positioner has bumped against an obstacle).

Abort On End Stop (reference mode `0b00001000`)

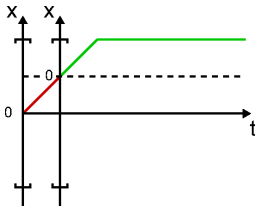


As described above, by default the positioner will start to look for a reference mark in the start direction and reverse the search direction if a physical end stop is detected.

If the abort on end stop flag is set then the positioner will *not* reverse the search direction on detecting a physical end stop. Instead it will stop and generate an error which means that the referencing is aborted and considered as failed.

This setting may be useful when it is necessary to forbid the movement of the positioner in a direction other than the initial search direction.

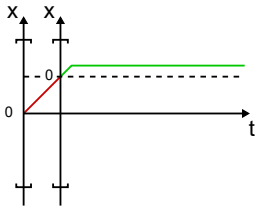
Continue On Reference Found (reference mode `0b00010000`)



Compared to the default referencing behavior this flag causes the positioner to continue to move in the current search direction after the reference has been found. The positioner does *not* stop or even turn around to return to the exact location of the reference mark. Instead the positioner must be stopped manually (or it is implicitly stopped by a physical end stop).

This setting may be useful e.g. when referencing several positioners synchronously that are mechanically connected in a parallel kinematic. A setup like this could cause one positioner to block and therefore fail to reference if another positioner has stopped because it has already found its reference mark.

Stop On Reference Found (reference mode `0b00100000`)



Compared to the default referencing behavior this flag causes the positioner to stop moving as soon as the reference has been found. The positioner does *not* turn around and return to the exact location of the reference mark. Instead the positioner simply stops where it is.

This implies that due to the small overshoot described above the positioner will not come to stop exactly on the reference mark. Since in these examples the zero position is on the reference mark, the position will not be zero after the referencing has completed.

2.7.3 Positioners With Multiple Reference Marks

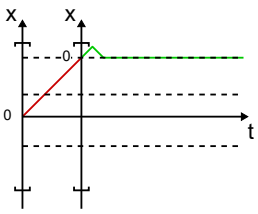
This section describes the behavior while referencing positioners with multiple reference marks in more detail. The general principle is that the positioner must pass over two adjacent reference marks. The physical position may then be determined by measuring the distance between these two marks. This method reduces the distance a positioner has to travel to determine its physical position compared to single reference marks, especially when operating with positioners with very long travel ranges.

As in the previous section the images on the right side illustrate the behavior of an example positioner that is being referenced. The vertical x-axis represents the travel range of the positioner. The square brackets indicate mechanical end stops. The dashed line indicates the positions of the reference marks.

In the examples the positioner always starts at position 0 and the physical position is unknown (red line). Once the reference mark has been found the physical position will become known (green line). The auto-zero flag is assumed to be set so the position will be set to zero once the physical position has been determined.

Default Behavior (with auto-zero, reference mode `0b00000100`)

By default the positioner will start to move in forward (positive) direction and look for a reference mark. When the positioner has found the first reference mark it will continue to move in forward direction and look for a second mark. As soon as the positioner has passed over the second reference mark the internal position will be updated (in this case set to 0 due to the auto-zero flag). This is indicated by the second x-axis having a different scale shift.

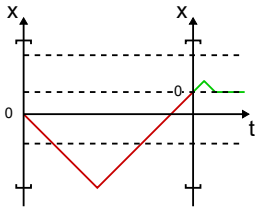


As in the previous examples the small overshoot represents the reaction time of the positioner stopping. The amount of the overshoot depends on factors like the velocity with which the referencing is performed, the mass that is mounted on the positioner or a possibly configured acceleration control (in which case it takes some time to decelerate the positioner).

The positioner will turn around and move to the exact location of the second reference mark. After this the referencing is complete.

Inverted Start Direction (with auto-zero, reference mode `0b00000101`)

Same as the default referencing, with the difference that the positioner will start to move in backward direction and look for two reference marks.

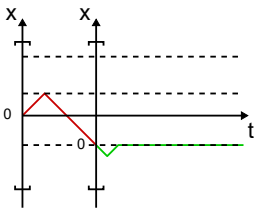


In this example the positioner passes over the first reference mark, but encounters a physical end stop before finding the second mark. The positioner will automatically reverse its search direction at the end stop and restart looking for a first reference mark.

As in the previous section please note that if the positioner should encounter a second end stop then the reference algorithm will be aborted. The positioner is stopped and an error event is generated. Reasons for this situation may be a mechanical or electrical defect (the controller does not register the reference signal for some reason) or the available travel range does not cover two reference marks (e.g. the positioner has bumped against an obstacle).

Reverse Direction (with auto-zero, reference mode `0b00000110`)

In this mode the positioner will start to move in forward (positive) direction and look for a reference mark. When the positioner has found the first reference mark it will *reverse* the movement direction and look for a second mark. As soon as the positioner has passed over the second reference mark the internal position will be updated (in this case set to 0 due to the auto-zero flag). This is indicated by the second x-axis having a different scale shift.



As in the previous examples the small overshoot represents the reaction time of the positioner stopping. The amount of the overshoot depends on factors like the velocity with which the referencing is performed, the mass that is mounted on the positioner or a possibly configured acceleration control (in which case it takes some time to decelerate the positioner).

The positioner will turn around and move to the exact location of the second reference mark. After this the referencing is complete.

This mode may further reduce the distance traveled by the positioner to determine its physical position.

2.7.4 Positioners With Endstop Reference

This section describes the behavior while referencing positioners with an endstop reference type in more detail. The general principle is to move the positioner towards one end of the travel range until a mechanical endstop is detected. The sensor signals are then used to align the position to the reference position with high repeat accuracy.

For these types of positioners the physical measuring scale is defined such that the zero position lies near the mechanical end stop that is used for referencing. Note that the scale therefore depends on the Safe Direction as well as the Logical Scale Inversion setting.

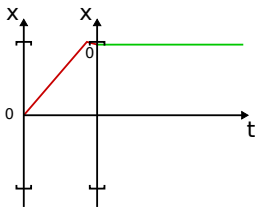
Positioners with an endstop reference type use the additional Safe Direction property to define the direction of the referencing movement instead of the start direction bit of the Referencing Options property.

All Referencing Options flags except the auto-zero flag are ignored when referencing towards an endstop.

NOTICE
Note that the end stop must be calibrated with `SA_CTL_Calibrate` before it can be properly used as a reference point.

Default Behavior (with auto-zero, reference mode `0b0000100`)

In this mode the positioner will start moving towards the configured Safe Direction and look for a mechanical end stop. In this example the Safe Direction is assumed to be set to forward (positive) direction.



Once the positioner has found the mechanical end stop it will move a short distance away from the end stop to find the exact reference using the position that was determined while calibrating the endstop.

As in the previous section the images on the right side illustrate the behavior of an example positioner that is being referenced. The vertical x-axis represents the travel range of the positioner. The square brackets indicate mechanical end stops.

In the examples the positioner starts at position 0 and the physical position is unknown (red line). Once the positioner is referenced the physical position will become known (green line). The auto-zero flag is assumed to be set so the position will be set to zero once the physical position has been determined.

2.7.5 Shifting the Measuring Scale

The *physical* measuring scale of a positioner is fix and cannot be changed. However, the MCS2 controller uses a *logical* measuring scale when calculating positions. The logical measuring scale may be shifted and/or inverted by the user so that the controller returns a desired position value at a certain physical position.

The relation between the physical and the logical scale is defined by two parameters. The *offset* value (which represents the shift) and the *inversion* value (which inverts the count direction) of the logical scale relative to the physical scale. The default value of the offset and the inversion is zero which makes the physical and the logical scale identical.

There are two methods to modify the offset value:

- Writing the Position property sets the offset implicitly by shifting the logical scale so that the current position equals the desired value.
- Writing the Logical Scale Offset property sets the offset explicitly and the current position will have a value that reflects the new scale shift.

The inversion value may be set by writing the Logical Scale Inversion property.

The offset and inversion values are stored in non-volatile memory. Once it is configured you only need to call the `SA_CTL_Reference` function to restore your settings on future power-ups.

Note: The behavior of the system when writing the Position property differs slightly depending on whether the physical position is known or not. When the physical position is unknown then writing the Position property will not update the scale offset value in the non-volatile memory. Likewise, writing the Logical Scale Offset property will have no immediate effect on the values read from the Position property. The following table summarizes the behavior.

	Physical position is known		Physical position is unknown	
	Set Position	Set Logical Scale	Set Position	Set Logical Scale
offset value is written to non-volatile memory	yes	yes	no	yes
function call has immediate effect on position values	yes	yes	yes	no

2.8 State Flags

2.8.1 Device State Flags

The device state may be read from the Device State property. The following flags are defined:

Bit	C-Definition	Mask
0	SA_CTL_DEV_STATE_BIT_HM_PRESENT	0x0001
1	SA_CTL_DEV_STATE_BIT_MOVEMENT_LOCKED	0x0002
8	SA_CTL_DEV_STATE_BIT_INTERNAL_COMM_FAILURE	0x0100
12	SA_CTL_DEV_STATE_BIT_IS_STREAMING	0x1000

HM Present (bit 0)

This flag indicates that a Hand Control Module is attached to the device.

Movement Locked (bit 1)

This flag indicates that the device is locked due to an emergency stop condition. (see section 2.17.2 "Emergency Stop Mode")

Internal Communication Failure (bit 8)

This flag indicates that an internal communication failure has occurred.

Is Streaming (bit 12)

This flag indicates that the device is currently performing a trajectory stream (see section 2.15 "Trajectory Streaming").

2.8.2 Module State Flags

The module state may be read from the Module State property. The following flags are defined:

Bit	C-Definition	Mask
0	SA_CTL_MOD_STATE_BIT_SM_PRESENT	0x0001
1	SA_CTL_MOD_STATE_BIT_BOOSTER_PRESENT	0x0002
2	SA_CTL_MOD_STATE_BIT_ADJUSTMENT_ACTIVE	0x0004
3	SA_CTL_MOD_STATE_BIT_IOM_PRESENT	0x0008
8	SA_CTL_MOD_STATE_BIT_INTERNAL_COMM_FAILURE	0x0100
12	SA_CTL_MOD_STATE_BIT_HIGH_VOLTAGE_FAILURE	0x1000
13	SA_CTL_MOD_STATE_BIT_HIGH_VOLTAGE_OVERLOAD	0x2000
14	SA_CTL_MOD_STATE_BIT_OVER_TEMPERATURE	0x4000

SM Present (bit 0)

This flag indicates whether a Sensor Module is currently attached to the Driver Module.

Booster Present (bit 1)

This flag indicates whether the Driver Module is equipped with a booster for high current signal output.

Adjustment Active (bit 2)

This flag indicates whether the module is performing an adjustment for the SmarAct PicoScale Laserinterferometer.

I/O Module Present (bit 3)

This flag indicates whether the Driver Module is equipped with an I/O Module.

Internal Communication Failure (bit 8)

This flag indicates that an internal communication error has occurred.

High Voltage Failure (bit 12)

This flag indicates that the module detected a power supply failure.

High Voltage Overload (bit 13)

This flag indicates that the module detected a power supply overload condition.

Over Temperature (bit 14)

This flag indicates that the module detected an over temperature condition.

2.8.3 Channel State Flags

The channel state may be read from the Channel State property. The following flags are defined:

Bit	C-Definition	Mask
0	SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING	0x0001
1	SA_CTL_CH_STATE_BIT_CLOSED_LOOP_ACTIVE	0x0002
2	SA_CTL_CH_STATE_BIT_CALIBRATING	0x0004
3	SA_CTL_CH_STATE_BIT_REFERENCING	0x0008
4	SA_CTL_CH_STATE_BIT_MOVE_DELAYED	0x0010
5	SA_CTL_CH_STATE_BIT_SENSOR_PRESENT	0x0020
6	SA_CTL_CH_STATE_BIT_IS_CALIBRATED	0x0040
7	SA_CTL_CH_STATE_BIT_IS_REFERENCED	0x0080
8	SA_CTL_CH_STATE_BIT_END_STOP_REACHED	0x0100
9	SA_CTL_CH_STATE_BIT_RANGE_LIMIT_REACHED	0x0200
10	SA_CTL_CH_STATE_BIT_FOLLOWING_LIMIT_REACHED	0x0400
11	SA_CTL_CH_STATE_BIT_MOVEMENT_FAILED	0x0800
12	SA_CTL_CH_STATE_BIT_IS_STREAMING	0x1000
14	SA_CTL_CH_STATE_BIT_OVER_TEMPERATURE	0x4000
15	SA_CTL_CH_STATE_BIT_REFERENCE_MARK	0x8000

Actively Moving (bit 0)

The channel is actively moving the positioner (open-loop or closed-loop).

Closed Loop Active (bit 1)

The channel is in closed-loop operation using sensor feedback (moving or holding the position).

Calibrating (bit 2)

The channel is busy performing a calibration sequence.

Referencing (bit 3)

The channel is busy performing a find reference sequence.

Move Delayed (bit 4)

The channel is waiting for the sensor to power-up before executing the movement command. This flag may be active if the sensor is operated in power save mode.

Sensor Present (bit 5)

A positioner with integrated sensor is attached to the channel. This indicates whether closed-loop movements may be performed.

Is Calibrated (bit 6)

The channel has valid signal correction calibration data for the configured positioner type. This flag is cleared when the positioner type is changed. It becomes one after a signal correction calibration sequence finished successfully.

Is Referenced (bit 7)

The channel "knows" its physical (absolute) position. After a power-up the physical position is unknown. After the reference mark has been found by calling `SA_CTL_Reference` the physical position becomes known.

End Stop Reached (bit 8)

The target position of a closed-loop movement command could not be reached because a mechanical end stop was detected. The positioner was stopped. The flag is cleared when a new movement command respectively stop command is issued.

Range Limit Reached (bit 9)

The positioner left the software configured range limit. The positioner was stopped. The flag is cleared when a new movement command respectively stop command is issued.

Following Limit Reached (bit 10)

The positioners following error exceeded the configured limit. The flag is cleared when a new movement command respectively stop command is issued.

Movement Failed (bit 11)

The last movement command failed. The Channel Error property may be read to determine the reason for the error.

Is Streaming (bit 12)

The channel is currently participating in a trajectory stream. As long as this flag is set the channel is unavailable for movement or configuration commands.

Over Temperature (bit 14)

The channel detected an over temperature condition.

Reference Mark (bit 15)

This flag reflects the state of the reference mark signal of the sensor.

2.9 Sensor Power Modes

In order for a positioner to track its position, its sensor needs to be supplied with power. However, since this generates heat (causing drift effects), it might be desirable to disable the sensors in some situations (especially in temperature critical environments). For this, there are three different modes of operation for the sensor, which may be configured individually for each channel with the Sensor Power Mode property. The following modes are available:

- **Disabled** In this mode the power supply of the sensor is turned off. This avoids the generation of heat by the sensor. Movement commands that require sensor feed back (such as closed-loop movements, referencing or calibrating) will not be executed. Instead, the generated `SA_CTL_EVENT_MOVEMENT_FINISHED` event holds an error code informing about the sensor state.
  Besides avoiding heat generation this mode may also be useful if the light that is emitted by the sensor interferes with other components of your setup (e.g. detectors inside an SEM chamber).
- **Enabled** In this mode the sensor is supplied with power continuously. All movement commands are executed normally.
- **PowerSave** If set to this mode the power supply of the sensor will be handled by the channel automatically. If the positioner is idle the sensor will be offline most of the time, avoiding unnecessary heat generation. A movement command (open-loop or closed-loop) will cause the channel to activate the sensor before the movement is started. Since it takes a few milliseconds to power-up the sensor, the movement will be delayed. The Channel State bit `SA_CTL_CH_STATE_BIT_MOVE_DELAYED` is set during this time.

Figure 2.5 illustrates the different sensor modes and shows when the sensor is supplied with power.

In this example the sensor mode is initially set to enabled. The sensor is continuously supplied with power. At time t_1 the sensor mode is switched to power save. In this mode the channel starts to pulse the power supply of the sensor to keep the heat generation low. At time t_2 a movement command is issued, which requires the sensor to be online in order to keep track of the current

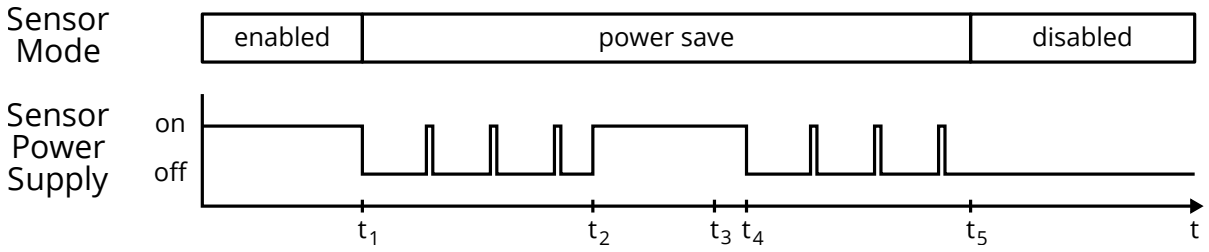


Figure 2.5: Sensor Modes

position. Note that the sensor mode stays unchanged during this time. After the movement has finished (t_3) an additional delay is started. While this delay the sensor stays online. (See the Sensor Power Save Delay property.) As soon as the delay time has elapsed (t_4) the channel will start to pulse the power supply again. At time t_5 the sensor mode is switched to disabled, in which the power supply is turned off continuously.

NOTICE

The sensor must be in *Enabled* or *PowerSave* mode for the sensor-present detection to be active. Accordingly, the `SA_CTL_CH_STATE_BIT_SENSOR_PRESENT` is not updated as long as the sensor is *Disabled*.

NOTICE

When in *PowerSave* or *Disabled* mode the positioner should not be moved by external means (e.g. by hand)! Since in these modes the power supply of the sensor is off most of the time or even continuously, the controller is not able to detect such movements. As a consequence the position data will become invalid. Furthermore, no error can be generated.

2.10 PicoScale Sensor Module

The MCS2 supports the SmarAct PicoScale laser interferometer as a high precision sensor module. This section explains the differences when using a PicoScale instead of the MCS2 sensor module. For a detailed description and setup of the PicoScale refer to the PicoScale User Manual.

For connecting the PicoScale to the MCS2 a special adapter cable (MCS2-A-PS-CABLE-1.5M-1.5M) is required. The adapter cable connects to the MCS2 and splits the high voltage output to three connections for positioners and forwards the data connection to the PicoScale.

When the PicoScale is connected to the MCS2, it is reported as a connected sensor module in the Module State Flags. Since the MCS2 only knows the sensor present flag in the Channel State Flags, but the PicoScale uses a number of different flags to indicate the system state, these flags are merged in the MCS2 context. For the sensor present flag to become active the following conditions must be met:

- System stable
- Channel enabled
- Channel data valid
- Beam not interrupted

For most of these flags to become active the channel needs to be adjusted. The adjustment can be performed using the PicoScale GUI or the MCS2 hand control module.

By default the MCS2 will use the PicoScale position data source as input for the control-loop. Alternatively, the calculation system can be selected as the input using the Sensor Input Select property. Note that the mapping between PicoScale calculation systems and MCS2 channels is static. The output of calculation system 0 of the PicoScale is used as input for channel 0 of the MCS2. Accordingly, calcSys 1 is used for channel 1 and calcSys 2 is used for channel 2.

When using the calculation system as input the conditions for the sensor present flag are as follows:

- System stable
- Calculation system data not interrupted

2.11 Following Error Detection

The following error detection feature may be used to inform the application if a commanded trajectory cannot be followed by a positioner precisely enough. The following error is, at a given time, the difference between the target position and the actual position while performing closed-loop movements. The positioner will always have a non-zero following error. The control-loop is tuned to reduce this error to its minimum. To enable the detection:

- The Following Error Limit property must be set to a non-zero value.
- The velocity control must be enabled (see Move Velocity).

The limit value is given in pm for linear positioners and in n° for rotary positioners. As soon as the configured limit is exceeded during a movement a `SA_CTL_EVENT_FOLLOWING_ERR_LIMIT` event is generated and the `SA_CTL_CH_STATE_BIT_FOLLOWING_LIMIT_REACHED` bit of the Channel State property will be set to one. The flag remains set until a new movement (or a `SA_CTL_Stop`) is commanded. Optionally the movement may be stopped automatically if the limit is exceeded. The `SA_CTL_POS_CTRL_OPT_BIT_STOP_ON_FOLLOWING_ERR` bit of the Positioner Control Options property must be set to one to stop the movement. In this case two events are generated. Firstly, the above mentioned `SA_CTL_EVENT_FOLLOWING_ERR_LIMIT`, secondly a `SA_CTL_EVENT_MOVEMENT_FINISHED` event. The latter will have its parameter set to `SA_CTL_ERROR_FOLLOWING_ERR_LIMIT`.

Note that the detection is not active during referencing movements.

2.12 Software Range Limit

While linear positioners have a limited physical travel range it might be useful to further limit this range if the positioner must not be allowed to move beyond a certain point. Rotary positioners usually have no physical end stops, but e.g. wiring may require to limit the rotation here as well. For these situations the MCS2 controller offers to limit the travel range of a positioner by software.

By default no range limit is set. To enable the range checks, the Range Limit Max property must be set to a higher value than the Range Limit Min property. Once the limits are set the positioner will not move beyond the boundaries of the range limit. This affects all movements except scan movements. If a movement command is issued that move the positioner beyond the defined limit then the positioner is stopped. A `SA_CTL_EVENT_MOVEMENT_FINISHED` event with its parameter set to `SA_CTL_ERROR_RANGE_LIMIT_REACHED` is generated and the Channel State bit `SA_CTL_CH_STATE_BIT_RANGE_LIMIT_REACHED` will be set to one. The flag remains set until a new movement (or a `SA_CTL_Stop`) is commanded. Further movements are only allowed if they move the positioner in the direction pointing back inside the range limit. This also applies if the positioner has been moved outside the defined range limit by external means.

Both the minimum and maximum position of the range limit behave similarly to a physical end stop. For example, the `SA_CTL_Reference` command will reverse its movement direction while looking for the reference mark if a range limit boundary is reached. If the reference mark is located outside the range limit then it will not be found.

Please note the following restrictions:

- The Range Limit Min and Range Limit Max properties are **not** saved to non-volatile memory and must be configured in each session.
- The range limits are **not** checked while performing the `SA_CTL_Calibrate` function for the signal correction calibration.
- The range limit has a limited accuracy. The positioner may pass over the boundary by a few micro meters resp. milli degrees. Therefore, the range should be defined with sufficient tolerance.

NOTICE

Setting the Position (as well as the Logical Scale Offset and Logical Scale Inversion properties) does **not** automatically adjust the software range limit accordingly. This means that shifting the measurement scale of the positioner with these commands will also shift the physical position of the software range limit. Therefore, care should be taken when working with these commands.

2.13 Stop Broadcasting

This feature can be used to broadcast a stop command to all channels on the MCS2 controller when a channel

- detects a mechanical end stop,
- reaches a software range limit (see section 2.12 "Software Range Limit") or
- exceeds a following error limit (see section 2.11 "Following Error Detection").

It is typically useful when multiple channels are moving simultaneously and one of the above conditions on one channel should cause a halt on all other channels. The channel that caused the broadcast stop generates a `SA_CTL_EVENT_MOVEMENT_FINISHED` event with its parameter holding the reason for the stop. (`SA_CTL_ERROR_END_STOP_REACHED`, `SA_CTL_ERROR_RANGE_LIMIT_REACHED` or `SA_CTL_ERROR_FOLLOWING_ERR_LIMIT`)

All other (currently moving) channels will be stopped and generate a `SA_CTL_EVENT_MOVEMENT_FINISHED` event with its parameter set to `SA_CTL_ERROR_ABORTED`.

NOTICE
A channel's behavior for a broadcast stop is the same as when executing a single `SA_CTL_Stop` command. Thus channels moving with acceleration control active may not come to halt immediately.

2.13.1 Stop Broadcast Configuration

The Broadcast Stop Options property defines the behavior of the broadcast stop feature. It holds a bit mask with the following mode bits:

Bit	Name	Short Description
0	End Stop Reached	Broadcast stop command if a mechanical end stop was detected.
1	Range Limit Reached	Broadcast stop command if a range limit was reached.
2	Following Limit Reached*	Broadcast stop command if a following error limit was exceeded.
3 .. 31	Reserved	These bits are reserved for future use. Should be set to zero.

*Note that the `SA_CTL_POS_CTRL_OPT_BIT_STOP_ON_FOLLOWING_ERR` bit of the Positioner Control Options property must be set to one to stop the movement and subsequently generate a broadcast stop on a following error limit.

Example

The example code below configures the device to issue a broadcast stop if channel 0 reaches an end stop or a Software Range Limit (±2mm).

```
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_RANGE_LIMIT_MIN, -2e9);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_RANGE_LIMIT_MAX, 2e9);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_BROADCAST_STOP_OPTIONS,
  (SA_CTL_STOP_OPT_BIT_END_STOP_REACHED |
   SA_CTL_STOP_OPT_BIT_RANGE_LIMIT_REACHED)
);
if (result) { /* handle error, abort */ }
```

2.14 Command Groups

When issuing movement or configuration commands they usually target a single channel of the device. However, when trying to move several channels synchronously communication delays induce a time offset of the resulting movements.

Command groups offer the possibility to define an atomic group of commands that is executed synchronously. In addition, a command group may not only be triggered via software, but alternatively via an external trigger.

To define a command group simply surround the commands that should be grouped with calls to the `SA_CTL_OpenCommandGroup` and `SA_CTL_CloseCommandGroup` functions and pass the transmit handle received from the `SA_CTL_OpenCommandGroup` function to all commands to be grouped.

For example, consider the code sequence below that configures two channels with the closed-loop absolute move mode and then moves both channels to some target position. (For simplicity the function return values are not handled in this example.)

```
SA_CTL_RequestWriteProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_MOVE_MODE,
  SA_CTL_MOVE_MODE_CL_ABSOLUTE,
  &rID0,
  0
);
SA_CTL_RequestWriteProperty_i32(
  dHandle,
```

```
  1,
  SA_CTL_PKEY_MOVE_MODE,
  SA_CTL_MOVE_MODE_CL_ABSOLUTE,
  &rID1,
  0
);
SA_CTL_Move(dHandle,0,1000000,0);
SA_CTL_Move(dHandle,1,2000000,0);
SA_CTL_WaitForWrite(dHandle,rID0);
SA_CTL_WaitForWrite(dHandle,rID1);
```

The next code snippet shows the same example, but the commands are put into a command group (changes are colored).

```
SA_CTL_TransmitHandle_t txHandle;
SA_CTL_OpenCommandGroup(dHandle,&txHandle
,SA_CTL_CMD_GROUP_TRIGGER_MODE_DIRECT);
SA_CTL_RequestWriteProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_MOVE_MODE,
  SA_CTL_MOVE_MODE_CL_ABSOLUTE,
  &rID0,
  txHandle
);
SA_CTL_RequestWriteProperty_i32(
  dHandle,
  1,
  SA_CTL_PKEY_MOVE_MODE,
  SA_CTL_MOVE_MODE_CL_ABSOLUTE,
  &rID1,
  txHandle
);
SA_CTL_Move(dHandle,0,1000000,txHandle);
SA_CTL_Move(dHandle,1,2000000,txHandle);
SA_CTL_CloseCommandGroup(dHandle,txHandle);
SA_CTL_WaitForWrite(dHandle,rID0);
SA_CTL_WaitForWrite(dHandle,rID1);
```

As a result the commands are treated as one command and the movements of both channels start synchronously (in this case as soon as the command group is closed, since the direct trigger mode is used).

One important thing to notice is that the `SA_CTL_WaitForWrite` function calls must be issued *after* the command group was closed. Otherwise the function calls will block. The same applies to commands that read properties from the device: put the `SA_CTL_RequestReadProperty` calls into the command group, but issue e.g. `SA_CTL_ReadProperty_i32` calls *after* the group close.

Note that synchronous property accesses cannot be put into a command group. Only the following *commands* may be added to command groups by passing the transmit handle to the function call:

- `SA_CTL_RequestReadProperty`

- SA_CTL_RequestWriteProperty_i32
- SA_CTL_RequestWriteProperty_i64
- SA_CTL_RequestWriteProperty_s
- SA_CTL_Calibrate
- SA_CTL_Reference
- SA_CTL_Move
- SA_CTL_Stop

In addition note that not all *properties* may be added to command groups. If a property is groupable or not is indicated in the detailed property description (See chapter 4 "Property Reference").

2.14.1 Command Groups vs. Output Buffer

Output buffer (as described in the High-Throughput Asynchronous Access for properties) are quite similar to command groups. However, there are still some differences which are outlined in the following.

- **Triggering** While output buffer are executed as soon as they are flushed, command groups may alternatively be triggered via an external trigger.
- **Size Limit** Command groups are somewhat limited in size regarding the number of commands that may be put into them. Output buffer are (theoretically) unlimited in size.
- **Atomicity** Output buffer simply try to optimize communication, but still treat the commands independently from each other. Output buffer are flushed on library level. In contrast, command groups optimize both communication and synchronized execution. They are flushed on controller level.

2.15 Trajectory Streaming

Trajectory streaming allows a multi DoF manipulator to follow specific trajectories using the MCS2 controller. All participating positioners are moved synchronously along the defined trajectory. This section describes the concepts of trajectory streaming and how an application program must use the API to perform a trajectory movement.

A trajectory movement requires special (user) software to pre-calculate support points of the trajectory (although the support points might also be calculated "on the fly"). These support points are then streamed to the controller which takes care of executing a synchronized movement of all participating channels.

2.15.1 General Streaming Concept

A trajectory stream is defined as a sequence of support points (*frames*). Each frame is a tuple of target positions for all channels that participate in the trajectory. Each target position in turn is a tuple of a channel index and a position value. Position values are given as a 64-bit integer value in little-endian format, representing pico meters for linear positioners and nano degrees for rotary positioners. All values are given as absolute (not relative) position values. Figure 2.6 shows the general format of a trajectory stream and figure 2.7 shows an example trajectory with the according binary stream data.

The timing with which the frames are executed is defined by the *stream rate* that is configurable by the user. This rate is constant for the duration of the stream.

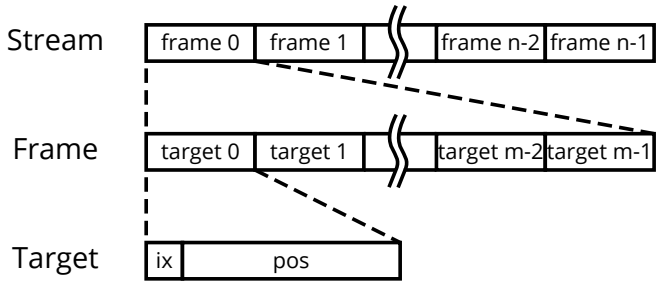


Figure 2.6: Trajectory Stream Format

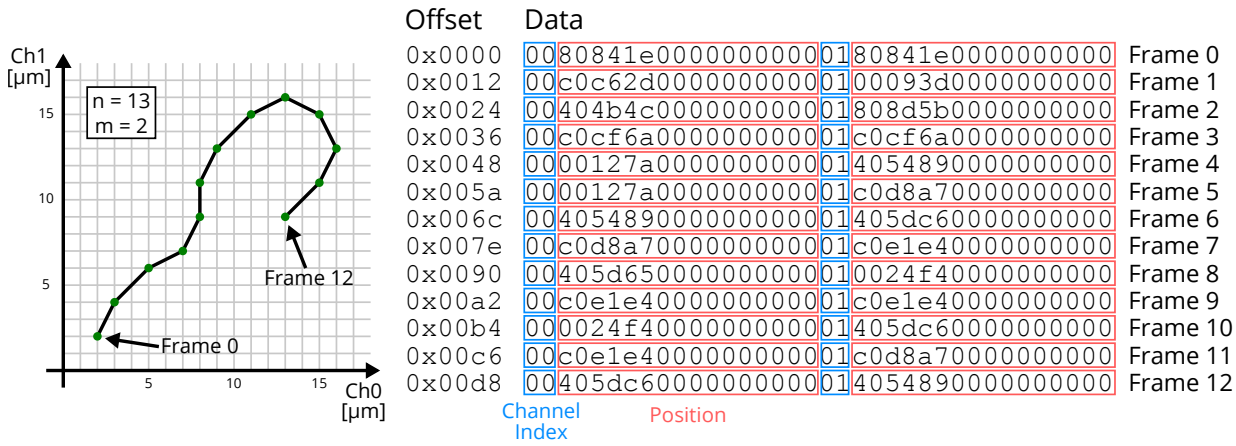


Figure 2.7: Trajectory Stream Example

Streaming Rules

When using trajectory streaming some rules must be heeded that are described in the following:

- Only one trajectory may be performed at a time. Suppose you have six channels available that are divided into two XYZ manipulators (A and B). Then you could start a trajectory with manipulator A. During this time it is not allowed to start a stream for manipulator B. If both manipulators are to be synchronized then the stream must contain all six channels from the beginning.

- The first frame of a stream defines which channels participate in the stream. All further frames must contain the same channels (in the same order). Otherwise a stream error is generated.
- A trajectory stream must consist of at least two frames (start frame and end frame).
- The movement between the support points is linearly interpolated by the controller (this is the default setting, see subsection 2.15.3). If a manipulator is supposed to perform an accelerated movement along the trajectory then the support points must be calculated accordingly.
- Channels that are not participating in the stream can still be fully controlled, while channels that are currently streamed may answer with a `SA_CTL_ERROR_BUSY_STREAMING` error code (see A.1) when sending certain configuration or movement commands.
- The sensors of all participating positioners must be enabled (in particular, the power save mode is not allowed, see section 2.9).

Flow Control

When the host transmits stream frames to the controller they are stored in a (FIFO) stream buffer in the controller. The controller then executes the buffered frames synchronously. While the frames are executed at a constant rate (the stream rate that the user has configured), the rate at which the controller receives frames from the host may vary. Typically the rate is considerably higher or frames arrive in bursts with intermissions (or both), e.g. due to USB / Ethernet latency or application interruption by the operating system (see figure 2.8).

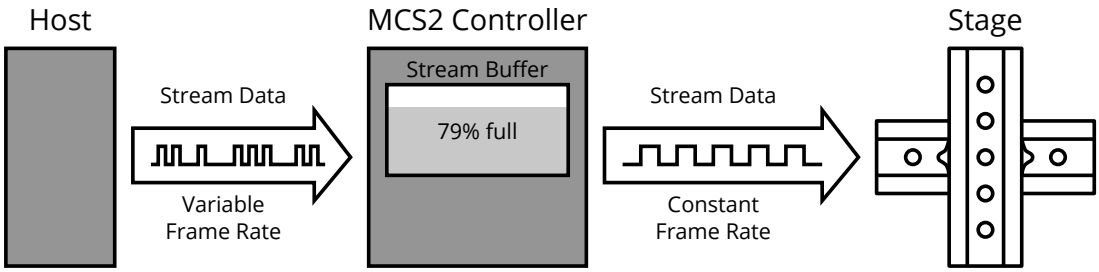


Figure 2.8: Flow Control

The library implements a flow control mechanism to prevent a buffer overflow on the controller:

- If the `SA_CTL_StreamFrame` function is called faster than the configured stream rate then the function may block from time to time, therefore implementing the flow control.
- If the `SA_CTL_StreamFrame` function is called slower than the configured stream rate then the streaming will eventually fail with a buffer underflow error.

The controller's stream buffer can hold up to 1024 tuples[1] and while it allows a synchronized and consistent stream, it also induces a delay to the incoming frames. This delay depends on the controller's buffer size, the number of channels that participate in the stream as well as the configured stream rate and can be determined by the following formula:

[1] A tuple consists of a target position and it's corresponding channel, see 2.15.1 (General Streaming Concept).

$$\textit{execution delay}\ [\text{s}] = \frac{\textit{buffer size}}{\textit{stream rate}\ [\text{Hz}] * \textit{number of stream channels}}$$

E.g. a stream with a frame containing three tuples (position data for three channels) and a configured stream rate of 1000 Hz would induce a constant buffer delay of $\frac{1024}{1000\ \text{Hz} * 3} = 0.341$ s.

2.15.2 Basic Approach

To execute a trajectory stream the following steps must be performed:

1. Configure the stream rate by writing the Stream Base Rate property (see section 4.9). This defines the rate (in Hz) with which the frames of the trajectory are executed.
2. Move all positioners that participate in the trajectory to their starting position (first frame of the stream). Otherwise starting the stream will likely cause unexpected behavior, since stream frames hold absolute position values and therefore the first frame could cause very high velocities that cannot be performed mechanically.
3. Open a stream by calling the `SA_CTL_OpenStream` function. It returns a stream handle that must be passed to the following function calls to associate them with the opened stream.
4. Supply the stream data by calling the `SA_CTL_StreamFrame` function once per frame that should be executed. Note: This function may block if the flow control needs to throttle the data rate. The function returns as soon as the frame was transmitted to the controller.
5. Close the stream by calling the `SA_CTL_CloseStream` function. To the controller this marks the end of the stream. If the stream is not closed properly with this function call (or aborted by calling `SA_CTL_AbortStream`) then the controller will generate a buffer underflow error after the last frame has been executed.

NOTICE

Behavioral differences when closing or aborting a stream:
As already described, all incoming frames are stored in an intermediate buffer by the device (see Flow Control). The basic approach, after having sent all frames to the device, is to call `SA_CTL_CloseStream`. This leads to execution of all pending frames and thus finishing the stream at the given position(s). If a stream is to be stopped immediately, the `SA_CTL_AbortStream` function can be used. This leads to a trajectory stop, while remaining frames already sent to the device are discarded.

2.15.3 Options

Before calling the `SA_CTL_OpenStream` function the Stream Options property can be configured to define the stream's behavior. This property holds a bit mask which is outlined in the following table.

Bit	Name	Short Description
0	Disable Linear Interpolation	Disable the linear interpolation between consecutive stream target positions.
1 .. 31	Reserved	These bits are reserved for future use.

Disable Linear Interpolation (streaming options `0x00` or `0x01`)

By default, the path between consecutive stream target positions is linearly interpolated. In some applications this behavior might be unwanted. The interpolation can therefore be disabled using this option, resulting in a point-to-point movement with the configured stream rate.

2.15.4 Trigger Modes

A trajectory stream may be configured to be triggered (started) by various events. For example, in some situations it can be useful to synchronize the stream rate of a trajectory with an external clock. A camera could then take snap shots with a frequency of 10 Hz while the stage moves along a trajectory with a time resolution of 200 Hz.

The desired trigger mode is passed to the `SA_CTL_OpenStream` function. The following trigger modes are available:

- `SA_CTL_STREAM_TRIGGER_MODE_DIRECT` (0)
- `SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_ONCE` (1)
- `SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_SYNC` (2)

Please note that in order to use the external trigger modes, the Input Trigger must be configured accordingly. Refer to section 2.17 "Input Trigger" on how to configure the device for triggered streaming.

Direct Mode

In this mode the stream is started as soon as the stream buffer on the controller contains enough data or has been closed (at which point a `SA_CTL_EVENT_STREAM_READY` event is generated).

External Once Mode

In this mode the stream is started by an external trigger that is fed into the device. Once the stream buffer on the controller contains enough data or has been closed a `SA_CTL_EVENT_STREAM_READY` event is generated to indicate that the stream is ready to be triggered by the external trigger. In this armed state the device waits for the trigger to occur and then generates a `SA_CTL_EVENT_STREAM_TRIGGERED` event. Further triggers are ignored in this mode.

External Synchronization Mode

This mode is used to synchronize the stream rate with an external clock which may be fed into the MCS2 controller. When the Stream External Sync Rate property is configured with the external clock rate then the trajectory stream will be synchronized with the external clock.

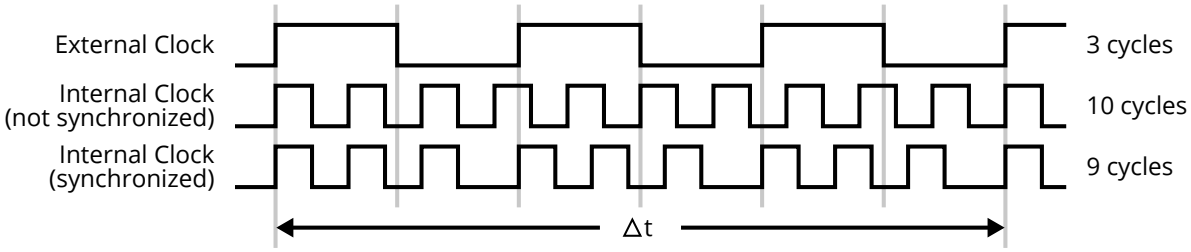


Figure 2.9: External synchronization with a 3:1 clock ratio

Figure 2.9 shows an example where the base stream rate is (should be) three times faster than the external sync rate (e.g. external 100 Hz, internal 300 Hz). The upper clock trace shows the external clock which makes 3 cycles within a given time window (*Δt*). The middle clock trace shows the internal clock while not being synchronized, being a speck too fast and making 10 cycles within the same time window. In the lower clock trace the internal clock is synchronized, making 9 clock cycles within the time window as desired. As a result the synchronization prevents the clocks from drifting apart.

NOTICE

The external synchronization feature has some restrictions that should be noted:

- In order to use the external synchronization feature the MCS2 controller must be equipped with an appropriate I/O Module.
- The Stream Base Rate must be a whole-number multiple of the external clock rate.
- The external clock rate may not be higher than the Stream Base Rate.

2.15.5 Stream Events

A trajectory stream that is started always generates the following events (in the order given):

1. `SA_CTL_EVENT_STREAM_READY` This event is generated as soon as the internal stream buffer of the device contains enough frames to start the stream without risking an immediate buffer underflow. The default buffer threshold is 50%. In case the stream is very short this event is generated as soon as the stream is closed.
2. `SA_CTL_EVENT_STREAM_TRIGGERED` This event is generated as soon as the device has started to execute the stream. In case of direct streaming the Stream Ready and the Stream Triggered events are generated at the same time. In case of externally triggered streaming the Stream Triggered event is delayed until the external trigger is detected which effectively starts the stream execution.

3. SA_CTL_EVENT_STREAM_FINISHED This event is generated when the stream has stopped executing. The event parameter indicates the result of the streaming. This could be a normal termination (SA_CTL_ERROR_NONE when executed to the last frame) or an error code specifying the reason for the abnormal termination.

2.15.6 Maximum Stream Rates

The maximum stable *stream rate* to be configured depends on the general communication load as well as the number of involved channels. The more channels are included in the trajectory stream, the higher the device's stream load. Table 2.2 shows possible stream rates for different number of streaming channels.

Channels	Stream Rate [Hz]	Channels	Stream Rate [Hz]	Channels	Stream Rate [Hz]
1	1000	7	480	13	260
2	1000	8	420	14	240
3	1000	9	370	15	220
4	840	10	340	16	210
5	670	11	300	17	200
6	560	12	280	18	190

Table 2.2: Stream Rate examples

For a more accurate determination of the maximum stream rate for the current setup the Stream Load Maximum property can be monitored while streaming. The property acts like a peak detector. The highest load level generated by the currently running stream is stored and may be read in percent with the Stream Load Maximum property. When starting the stream the load value is reset to zero.

It is recommended to configure the trajectory stream (e.g. the Stream Base Rate) with some headroom to the maximum load to guarantee a stable operation. If an overload is detected the trajectory stream aborts with an SA_CTL_ERROR_SYNC_FAILED error.

Note that channels which are not part of the current stream can be fully controlled while a stream is running. However, doing so always generates some peak load which must be considered. Note further that streaming to multiple channels with high stream rates may also affect the performance for operations concerning other channels.

2.16 Auxiliary Inputs and Outputs

The MCS2 device offers auxiliary inputs and outputs to interface to external equipment.

NOTICE

The device must be equipped with an additional I/O module to use auxiliary inputs and outputs. The characteristics as well as the number of inputs and outputs vary depending on the specific type of I/O module. Please refer to the MCS2 User Manual for detailed electrical specifications.

2.16.1 Digital Device Input

Digital device inputs allow to synchronize movements to external events. Synchronizing the trajectory streaming or triggering command groups as well as aborting movements by triggering an emergency stop is possible. This feature is called "Input Trigger". See section 2.17 "Input Trigger" for the configuration of the input trigger.

2.16.2 Fast Digital Outputs

Fast digital outputs may be used to trigger external equipment like detectors or cameras depending on the current position of a positioner. This feature is called "Output Trigger". See section 2.18 "Output Trigger" for the configuration of the output trigger.

2.16.3 General Purpose Digital Inputs/Outputs

General purpose digital inputs and outputs may be used to control lights, relays, dispensers, etc. or to read the state of safety switches, light barriers, etc.

Digital Inputs

The Aux Digital Input Value property may be used to read the digital inputs of an I/O module. The first bit (bit 0) of the input value corresponds to the first digital input (GP-DIN-1), the second bit (bit 1) corresponds to the second input (GP-DIN-2) and so on.

It is possible to enable an event notification for the digital inputs to be notified if an input changes. Thus, continuous polling of the Aux Digital Input Value property can be avoided. To enable the event set the `SA_CTL_IO_MODULE_OPT_BIT_EVENTS_ENABLED` bit of the I/O Module Options property to one. Whenever a change of one or more of the general purpose digital inputs happens the device generates a `SA_CTL_EVENT_DIGITAL_INPUT_CHANGED` event with its parameter holding the new state of the inputs. Note that the input state capture frequency for the event generation is limited to approx. 100 Hz. See section 2.4 "Event Notifications" for more information on receiving events.

Example:

```
SA_CTL_Result_t result;
// read the digital inputs
int32_t input;
result = SA_CTL_GetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_DIGITAL_INPUT_VALUE, &input
);
if (result == SA_CTL_ERROR_NONE) {
  // 'input' holds the value of the digital inputs
}
// enable the digital input changed event
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_IO_MODULE_OPTIONS,
    SA_CTL_IO_MODULE_OPT_BIT_EVENTS_ENABLED
);
// -> receive event using the SA_CTL_WaitForEvent() function
```

Digital Outputs

NOTICE

The digital output driver circuit is disabled by default and must be enabled by setting the `SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUT_ENABLED` bit of the I/O Module Options property.

The following properties may be used to modify the digital outputs:

- The **Aux Digital Output Value** property sets all outputs at once to a defined value.
- The **Aux Digital Output Set** property sets all specified outputs to one without modifying the other ones.
- The **Aux Digital Output Clear** property clears all specified outputs without modifying the other ones.

The first bit (bit 0) of the output value corresponds to the first digital output (GP-DOUT-1), the second bit (bit 1) corresponds to the second output (GP-DOUT-2) and so on. Note that the general purpose outputs are designed as open-collector outputs. This means that the output logic is inverted. Writing a one to an output switches the output transistor on which leads to a low signal level at the output pin. The following code shows how to modify digital outputs of an I/O module:

```
SA_CTL_Result_t result;
// set the output driver voltage level to 5V
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_IO_MODULE_VOLTAGE, SA_CTL_IO_MODULE_VOLTAGE_5V
);
if (result) { /* handle error, abort */ }
// enable the digital output driver circuit of the I/O module
result = SA_CTL_SetProperty_i32(dHandle,0,
```

```
  SA_CTL_PKEY_IO_MODULE_OPTIONS,
  SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUTS_ENABLED
);
if (result) { /* handle error, abort */ }
// first set all digital outputs of the I/O module to a specific value
// note: electrical levels are inverted due to the open-collector outputs
// DOUT-4 | DOUT-3 | DOUT-2 | DOUT-1 |
// H(1)   | L(0)   | H(1)   | L(0)   |
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_VALUE, 0x00000005
);
if (result) { /* handle error, abort */ }
// next set output 2 (DOUT-2) without modifying the other outputs
// DOUT-4 | DOUT-3 | DOUT-2 | DOUT-1 |
// H(1)   | L(0)   | L(0)   | L(0)   |
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_SET, 0x00000002
);
if (result) { /* handle error, abort */ }
// last clear output 1 (DOUT-1) without modifying the other outputs
// DOUT-4 | DOUT-3 | DOUT-2 | DOUT-1 |
// H(1)   | L(0)   | L(0)   | H(1)   |
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_CLEAR, 0x00000001
);
```

2.16.4 Fast Analog Inputs

Fast analog inputs may be used to read analog voltage signals. An application can poll the Aux I/O Module Input0 / Input1 Value properties and use the data for further processing. The I/O module has a total number of six analog inputs which are mapped in groups of two to the channels of the corresponding driver module. The following table shows the combinations of channel index and property which must be used to read the input values of the six analog inputs:

Analog Input	Channel Index	Property
AIN-1	0	SA_CTL_PKEY_AUX_IO_MODULE_INPUT0_VALUE
AIN-2	1	SA_CTL_PKEY_AUX_IO_MODULE_INPUT0_VALUE
AIN-3	2	SA_CTL_PKEY_AUX_IO_MODULE_INPUT0_VALUE
AIN-4	0	SA_CTL_PKEY_AUX_IO_MODULE_INPUT1_VALUE
AIN-5	1	SA_CTL_PKEY_AUX_IO_MODULE_INPUT1_VALUE
AIN-6	2	SA_CTL_PKEY_AUX_IO_MODULE_INPUT1_VALUE

The following code shows how to read the first analog input assigned to the second channel (channel index 1) of a device (AIN-2):

```
SA_CTL_Result_t result;
int64_t input;
result = SA_CTL_GetProperty_i64(dHandle,1,
  SA_CTL_PKEY_AUX_IO_MODULE_INPUT0_VALUE, &input
);
if (result == SA_CTL_ERROR_NONE) {
  // 'input' holds the value of the analog input AIN-2
}
```

2.16.5 Using Analog Inputs as Control-Loop Feedback

The MCS2 supports to feed external analog signals into the control-loop of a channel. This allows to implement applications like aligning a sample depending on the light intensity of an external light detector or force feedback for a gripper, etc. These tasks require a more complex configuration which is described in the following.

Note that the total number of six analog inputs of the I/O module are mapped in groups of two to the channels of the corresponding driver module. This means that per channel only two of the analog inputs may be used as control-loop feedback. (See Aux I/O Module Input Index property).

CAUTION

It is the user's responsibility to guarantee that a valid signal is fed into the input and that all properties (input ranges, PID parameters, etc.) are configured to reasonable values before enabling the closed-loop operation. Configuring inappropriate values may result in unstable or unexpected behavior of the positioners and potential damage of the stage.

To use an auxiliary input as control-loop feedback the following properties must be configured:

- The actual analog input must be selected with the **Aux Input Select** and **Aux I/O Module Input Index** properties.
- The analog input range must be selected with the **I/O Module Analog Input Range** property.
- The **Aux Positioner Type** must be set to a custom positioner type slot. This slot must be configured with a set of PID parameters with the Tuning and Customizing Properties. Note that not all positioner type properties have a meaning when used as auxiliary positioner type. The following properties are of interest to configure the PID loop: Positioner P Gain, Positioner I Gain, Positioner D Gain, Positioner Anti Windup, Positioner PID Shift.
- Depending on the specific application and the type of feedback signal it may be necessary to disable the endstop detection by setting the **Positioner ESD Distance Threshold** property to zero. Whenever the auxiliary input value represents a set-point for the control-loop instead of a current position of the positioner the endstop detection must be disabled. (E.g. a force signal in a force-feedback-gripper application defines the set-point and does not follow the actual position.)

- The modifications should be saved to a custom positioner type slot with the **Save Positioner Type** property.
- The direction sense of the feedback must be defined with the **Aux Direction Inversion** property. It must match the direction sense of the control-loop output. Otherwise a runaway condition may occur when commanding a closed-loop movement.
- The **Control Loop Input** property must be set to `SA_CTL_CONTROL_LOOP_INPUT_AUX_IN` to feed the auxiliary input signal into the PID controller.

Using an auxiliary input as control-loop feedback has some special characteristics which need to be considered:

- The `SA_CTL_CH_STATE_BIT_SENSOR_PRESENT` flag of the Channel State refers to the position control-loop input. The auxiliary input signal is always treated as 'present' for the control-loop.
- The auxiliary input value is reflected in the 'current position' of a channel, even if the representation of the input signal has a physical unit different from 'position'. Commanding the channels 'target position' with the `SA_CTL_Move` function always refers to the absolute value and range of the input signal.
- The auxiliary input signal is defined as absolute value, thus it is not possible to define a logical scale offset, e.g. by setting the position with the Position property. Doing so affects the position calculation of an integrated sensor of a positioner (if there is one). Several properties give access to the position of an integrated sensor as well as the auxiliary input values regardless of the actual signal currently used as feedback signal. Refer to figure 2.10 for the different signal paths and properties in this context.
- Two positioner type slots are used to define the tuning parameters of the control-loop:
  - The Aux Positioner Type property defines a set of tuning parameters which is used if an auxiliary input provides the control-loop feedback.
  - The Positioner Type property defines the parameters for all other configurations.

  The corresponding set of parameters is configured implicitly when changing the control-loop input. This allows to switch between two operation modes without manually reconfiguring the control-loop tuning.

The following figure shows the auxiliary input configuration for each channel:

2.16.6 Analog Outputs

Analog outputs generate analog voltage control signals for external amplifiers, dispensers etc.

NOTICE

The analog output driver circuit is in a high-impedance state by default and must be enabled by setting the `SA_CTL_IO_MODULE_OPT_BIT_ANALOG_OUTPUT_ENABLED` bit of the I/O Module Options property.

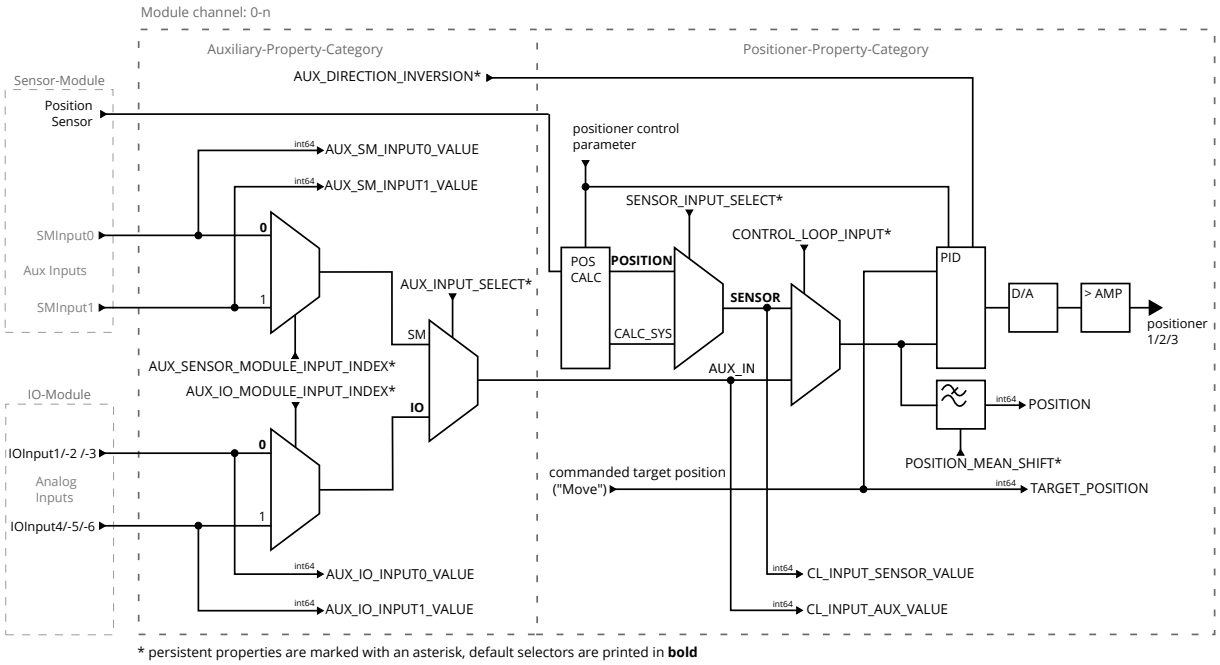


Figure 2.10: Auxiliary Input Configuration (per channel)

The Aux Analog Output Value0 / Value1 properties may be used to output an analog voltage on the I/O module analog outputs (AOUT-1 and AOUT-2).

The following code shows how to set both analog outputs of an I/O module:

```
SA_CTL_Result_t result;
// set the output value of analog output0 (AOUT-1) to zero
// which corresponds to 0V
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_ANALOG_OUTPUT_VALUE0, 0
);
if (result) { /* handle error, abort */ }
// set the output value of analog output1 (AOUT-2) to max
// which corresponds to +10V
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_ANALOG_OUTPUT_VALUE1, 32768
);
if (result) { /* handle error, abort */ }
```

2.17 Input Trigger

Digital input triggers allow to synchronize the device to external clock signals or events. The input trigger may be used as an emergency stop input, to synchronize the trajectory streaming or to trigger command groups (e.g. a group of movement commands).

NOTICE
In order to use the input trigger the device must be equipped with an additional I/O module.

The following properties may be used to configure the input trigger:

- The **Device Input Trigger Mode** property defines how the device reacts to incoming trigger signals. The available trigger modes are described in more detail in the following sections.
- The **Device Input Trigger Condition** property defines whether to react to rising or falling edges.

2.17.1 Disabled Mode

This is the default mode in which all activities on the input line are ignored.

2.17.2 Emergency Stop Mode

The emergency stop input trigger mode allows to use the input trigger to issue an emergency stop. In terms of the MCS2 an emergency stop stops all active movements. More precisely, the device will hard-stop all channels and aborts active streams and command groups. Note that channels moving with acceleration control active will also be stopped immediately. The desired behavior how to handle the emergency stop situations can further be configured by setting the Emergency Stop Mode property to one of the following modes:

SA_CTL_EMERGENCY_STOP_MODE_NORMAL This is the default mode. In this mode the configured input trigger condition issues an emergency stop. After such an event the device continues to behave normally.

SA_CTL_EMERGENCY_STOP_MODE_RESTRICTED In this mode the configured input trigger condition will issue an emergency stop and make the device enter a locked state. In this state you may communicate with the device normally, but all movement commands will return an SA_CTL_ERROR_MOVEMENT_LOCKED error. The locked state may be reset by setting the emergency stop mode to any valid value, thereby unlocking the movement again.

SA_CTL_EMERGENCY_STOP_MODE_AUTO_RELEASE In this mode the configured input trigger condition will issue an emergency stop and make the device enter a locked state. In this state you may communicate with the device normally, but all movement commands will return an SA_CTL_ERROR_MOVEMENT_LOCKED error. This state remains until either the emergency stop mode is set to any valid value or the input trigger line is released (inverse edge is detected).

The following code gives an example for the configuration of the input trigger when used as emergency stop. After a successful configuration a falling edge on the input trigger will issue an emergency stop. The following behavior is defined by the configured emergency stop mode (in this case the device continues normally).

```
SA_CTL_Result_t result;
// set input trigger mode to emergency stop
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_MODE,
  SA_CTL_DEV_INPUT_TRIG_MODE_EMERGENCY_STOP
);
if (result) { /* handle error, abort */ }
// set input trigger condition to falling edge
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_CONDITION,
  SA_CTL_TRIGGER_CONDITION_FALLING
);
if (result) { /* handle error, abort */ }
// configure emergency stop mode
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_EMERGENCY_STOP_MODE,
  SA_CTL_EMERGENCY_STOP_MODE_NORMAL
);
if (result) { /* handle error, abort */ }
```

2.17.3 Stream Sync Mode

The stream sync input trigger mode allows to use the streaming's external trigger modes. Calling `SA_CTL_OpenStream` with either `SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_ONCE` or `SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_SYNC` will start resp. synchronize the stream to the input trigger. See section 2.15 "Trajectory Streaming" for more information.

The following code gives an example for the configuration of the input trigger when used to start the stream. After a successful configuration a stream is opened with trigger mode *external once* parameter. If the stream is ready (stream ready event received), a rising edge on the input trigger will start the trajectory's execution.

```
SA_CTL_StreamHandle_t sHandle;
SA_CTL_Result_t result;
// set input trigger mode to stream sync
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_MODE,
  SA_CTL_DEV_INPUT_TRIG_MODE_STREAM
);
if (result) { /* handle error, abort */ }
// set input trigger condition to rising edge
```

```
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_CONDITION,
  SA_CTL_TRIGGER_CONDITION_RISING
);
if (result) { /* handle error, abort */ }
// open stream with trigger mode external once
result = SA_CTL_OpenStream(
  dHandle,
  &sHandle,
  SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_ONCE
);
if (result) { /* handle error, abort */ }
// ...
// start streaming frames to the device
// ...
// >> stream ready event <<
// device is now waiting for the external trigger condition to start
// the stream
```

2.17.4 Command Group Sync Mode

The command group sync input trigger mode allows to use the command groups external trigger mode. Calling `SA_CTL_OpenCommandGroup` with the trigger mode `SA_CTL_CMD_GROUP_TRIGGER_MODE_EXTERNAL` will then delay the groups execution until the external input trigger occurs. See section 2.14 "Command Groups" for more information.

The following code gives an example for the configuration of the input trigger when used for starting command groups. After a successful configuration of the input trigger a command group is opened with the external trigger mode parameter, filled (e.g. with `SA_CTL_Move` commands) and then closed. The groups execution though is delayed until the device detects a rising edge on the input trigger.

```
SA_CTL_TransmitHandle_t tHandle;
SA_CTL_Result_t result;
// set input trigger mode to cmd group sync
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_MODE,
  SA_CTL_DEV_INPUT_TRIG_MODE_CMD_GROUP
);
if (result) { /* handle error, abort */ }
// set input trigger condition to rising edge
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_CONDITION,
```

```
  SA_CTL_TRIGGER_CONDITION_RISING
);
if (result) { /* handle error, abort */ }
// open command group with trigger mode external
result = SA_CTL_OpenCommandGroup(
  dHandle,
  &tHandle,
  SA_CTL_CMD_GROUP_TRIGGER_MODE_EXTERNAL
);
if (result) { /* handle error, abort */ }
// ...
// fill command group
// ...
// close command group
result = SA_CTL_CloseCommandGroup(dHandle, tHandle);
if (result) { /* handle error, abort */ }
// command group is now waiting for the external trigger condition
```

2.17.5 Event Trigger Mode

The event input trigger mode allows to get a notification whenever an electrical trigger signal was detected on the trigger input. This mode is useful to simply inform the software about the occurrence of an external trigger signal without any further actions on the controller.

Note that the maximum frequency of the input signal should be limited to 500 Hz in this mode.

The following code gives an example for the configuration of the input trigger when used to get event notifications. After a successful configuration a rising edge on the input trigger will generate an external input triggered event.

```
SA_CTL_Result_t result;
// set input trigger mode to event trigger
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_MODE,
  SA_CTL_DEV_INPUT_TRIG_MODE_EVENT
);
if (result) { /* handle error, abort */ }
// set input trigger condition to rising edge
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_DEV_INPUT_TRIG_CONDITION,
  SA_CTL_TRIGGER_CONDITION_RISING
);
if (result) { /* handle error, abort */ }
// wait for events
SA_CTL_Event_t event;
result = SA_CTL_WaitForEvent(dHandle,&event,SA_CTL_INFINITE);
```

```
if (result) { /* handle error, abort */ }
// ...
```

2.18 Output Trigger

In some applications it is useful to have the controller output a trigger signal each time the position of a channel has made a certain increment or the target position has been reached. The trigger signals may then be used by external logic (e.g. to trigger a camera).

NOTICE
In order to use the output trigger signals the device must be equipped with an additional I/O module. Since each I/O module is connected to a specific driver module the output trigger signals are assigned to the channels of the corresponding driver module.

The following properties may be used to configure the output trigger:

- The **Channel Output Trigger Mode** property defines what is output to the corresponding output pin. The available trigger modes are described in more detail in the following sections.
- The **Channel Output Trigger Polarity** property defines the polarity of the output trigger signal.
- The **Channel Output Trigger Pulse Width** property specifies the pulse width of a trigger output pulse.
- The **I/O Module Options** property bit `SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUT_ENABLED` must be set to enable the output driver circuit.
- The **I/O Module Voltage** selects the output voltage of the pin.

Note that the I/O module settings are global for all output channels of the I/O module. The following example code enables the output trigger and configures the output voltage to 5 V.

```
SA_CTL_Result_t result;
// set the output driver voltage level to 5V
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_IO_MODULE_VOLTAGE,
  SA_CTL_IO_MODULE_VOLTAGE_5V
);
if (result) { /* handle error, abort */ }
// enable the output driver circuit of the I/O module
```

```
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_IO_MODULE_OPTIONS,
  SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUT_ENABLED
);
if (result) { /* handle error, abort */ }
```

2.18.1 Constant Mode

This is the default mode in which a constant level is output. The level corresponds to the inactive state of the configured Channel Output Trigger Polarity.

The following example shows how user defined levels can be output in this mode.

```
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_i32(
  dHandle,
  2,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY,
  SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle,
  2,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_MODE,
  SA_CTL_CH_OUTPUT_TRIG_MODE_CONSTANT
);
if (result) { /* handle error, abort */ }
// output of channel 2 level is now low
// perform some tasks...
result = SA_CTL_SetProperty_i32(
  dHandle,
  2,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY,
  SA_CTL_TRIGGER_POLARITY_ACTIVE_LOW
);
if (result) { /* handle error, abort */ }
// output of channel 2 level is now high
```

2.18.2 Position Compare Mode

The position compare mode allows to generate trigger signals according to the current position of a positioner. One independent trigger per channel is available.

The Channel Position Compare Limit Min and Channel Position Compare Limit Max properties are used to define the working range for the trigger generation. This is especially useful to implement

raster scanning applications where e.g. an X/Y stage moves a sample along a specific trajectory and a detector must be triggered according to the current position of a sample.

Once a limit was passed by the positioner the direction of the position increment is reverted or reset to the starting threshold to define the next trigger position. If the Channel Position Compare Direction is set to `SA_CTL_EITHER_DIRECTION` the increment is reversed to continuously generate pulses on the way back to the starting position. This is known as 'snake scanning'. Otherwise the increment is reset to the defined start threshold to restart the pulse generation after the positioner was moved back and reversed its movement direction. This is known as 'line scanning'.

Note that the reversal positions of the movement trajectory should be defined with sufficient tolerance to reliably pass the limits while moving.

The following code gives an example for the configuration of the output trigger for channel 1. The movement is commanded with its reversal points defined to 0 and 5 mm. After enabling the trigger the channel will generate a 1 µs pulse (0.5 µs high, 0.5 µs low) once the position of channel 1 passed 1 mm in forward direction. Furthermore every 100 µm consecutive pulses are output until the max limit of 4.5 mm was passed. This is repeated for every movement starting from zero position.

```
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_i64(
  dHandle, 1,
  SA_CTL_PKEY_CH_POS_COMP_START_THRESHOLD, 1e9
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i64(
  dHandle, 1,
  SA_CTL_PKEY_CH_POS_COMP_INCREMENT, 100e6
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle, 1,
  SA_CTL_PKEY_CH_POS_COMP_DIRECTION,
  SA_CTL_FORWARD_DIRECTION
);
if (result) { /* handle error, abort */ }

result = SA_CTL_SetProperty_i64(
  dHandle, 1,
  SA_CTL_PKEY_CH_POS_COMP_LIMIT_MIN, 500e6
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i64(
  dHandle, 1,
  SA_CTL_PKEY_CH_POS_COMP_LIMIT_MAX, 4500e6
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle, 1,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY,
  SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH
```

```
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle, 1,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_PULSE_WIDTH, 1000
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle, 1,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_MODE,
  SA_CTL_CH_OUTPUT_TRIG_MODE_POSITION_COMPARE
);
if (result) { /* handle error, abort */ }
// start movement between position 0 and 5mm
```

2.18.3 Target Reached Mode

The target reached mode allows to generate a pulse once a closed-loop movement command finished and the positioner reached its target position. The pulse is only generated for successfully finished movement commands.

The following code gives an example for the configuration of the target reached output trigger for channel 1. After enabling the trigger the output of the channel will generate a pulse of defined length once the target position of a movement has been reached.

```
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_i32(
  dHandle,
  1,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY,
  SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle,
  1,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_PULSE_WIDTH,
  1000
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle,
  1,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_MODE,
  SA_CTL_CH_OUTPUT_TRIG_MODE_TARGET_REACHED
);
if (result) { /* handle error, abort */ }
```

2.18.4 Actively Moving Mode

The actively moving mode generates an output level similar to the actively moving Channel State bit. The output level is in the active state while the positioner is moving and inactive otherwise.

The following example code configures channel 2 to output a high level while the positioner is moving.

```
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_i32(
  dHandle,
  2,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY,
  SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH
);
if (result) { /* handle error, abort */ }
result = SA_CTL_SetProperty_i32(
  dHandle,
  2,
  SA_CTL_PKEY_CH_OUTPUT_TRIG_MODE,
  SA_CTL_CH_OUTPUT_TRIG_MODE_ACTIVELY_MOVING
);
if (result) { /* handle error, abort */ }
```

2.19 Feature Permissions

The MCS2 has a feature permission system which allows to activate special features via an software activation process without physically returning the controller to SmarAct. New features may be unlocked by upgrading the controller with an upgrade file. The MCS2 Service Tool is used to perform this upgrade. Please contact SmarAct for the details on purchasing a feature upgrade.

Currently the following features are available:

- Low Vibration Actuator Mode (Actuator Mode property)
- Advanced Sensor Correction (Signal Correction Options property)

In case that a feature is not activated on a controller, trying to enable it will generate a `SA_CTL_ERROR_PERMISSION_DENIED` error.

3 *FUNCTION REFERENCE*

3.1 Function Summary

Table 3.1 – Function Summary

Function Name	Short Description	Page
SA_CTL_GetFullVersionString	Returns the version of the library as a human readable string.	78
SA_CTL_GetResultInfo	Returns a human readable error string for the given error code.	79
SA_CTL_GetEventInfo	Returns a human readable info string for the given event.	80
SA_CTL_FindDevices	Returns a list of locator strings of connected devices.	81
SA_CTL_Open	Opens a connection to a device.	83
SA_CTL_Close	Closes a connection to a device.	84
SA_CTL_Cancel	Unblocks all blocking API calls.	85
SA_CTL_GetProperty_i32	Directly returns the value of a 32 bit integer property.	86
SA_CTL_SetProperty_i32	Directly sets the value of a 32 bit integer property.	88
SA_CTL_SetPropertyArray_i32	Directly sets the value of a 32 bit integer array property.	89
SA_CTL_GetProperty_i64	Directly returns the value of a 64 bit integer property.	90
SA_CTL_SetProperty_i64	Directly sets the value of a 64 bit integer property.	91
SA_CTL_SetPropertyArray_i64	Directly sets the value of a 64 bit integer array property.	92
SA_CTL_GetProperty_s	Directly returns the value of a string property.	93
SA_CTL_SetProperty_s	Directly sets the value of a string property.	95

Continued on next page

Table 3.1 – *Continued from previous page*

Function Name	Short Description	Page
SA_CTL_RequestReadProperty	Requests the value of a property (non-blocking).	96
SA_CTL_ReadProperty_i32	Reads the value of a requested 32 bit integer property.	98
SA_CTL_ReadProperty_i64	Reads the value of a requested 64 bit integer property.	99
SA_CTL_ReadProperty_s	Reads the value of a requested string property.	100
SA_CTL_RequestWriteProperty_i32	Requests to write the value of a 32 bit integer property (non-blocking).	102
SA_CTL_RequestWriteProperty_i64	Requests to write the value of a 64 bit integer property (non-blocking).	104
SA_CTL_RequestWriteProperty_s	Requests to write the value of a string property (non-blocking).	105
SA_CTL_RequestWritePropertyArray_i32	Requests to write the value of a 32 bit integer array property (non-blocking).	106
SA_CTL_RequestWritePropertyArray_i64	Requests to write the value of a 64 bit integer array property (non-blocking).	107
SA_CTL_WaitForWrite	Waits until a write operation has finished.	108
SA_CTL_CancelRequest	Cancels a non-blocking read or write request.	109
SA_CTL_CreateOutputBuffer	Opens up an output buffer for delayed transmission of several commands.	110
SA_CTL_FlushOutputBuffer	Flushes an output buffer and triggers the transmission to the device.	111
SA_CTL_CancelOutputBuffer	Cancels an output buffer and discards buffered commands.	112
SA_CTL_OpenCommandGroup	Opens up an atomic command group.	113
SA_CTL_CloseCommandGroup	Flushes a command group and makes all commands of the group take effect.	114
SA_CTL_CancelCommandGroup	Cancels a command group and discards buffered commands.	115
SA_CTL_WaitForEvent	Listens to events from the device.	116
SA_CTL_Calibrate	Performs a calibration.	118
SA_CTL_Reference	Performs a finding of a reference mark.	120

Continued on next page

Table 3.1 – *Continued from previous page*

Function Name	Short Description	Page
SA_CTL_Move	Performs a movement.	122
SA_CTL_Stop	Aborts all ongoing movements.	124
SA_CTL_OpenStream	Opens a stream.	125
SA_CTL_StreamFrame	Sends a previously assembled frame to the device.	127
SA_CTL_CloseStream	Closes a stream.	129
SA_CTL_AbortStream	Aborts a stream.	131

3.2 Detailed Function Description

3.2.1 SA_CTL_GetFullVersionString

Interface:

```
const char* SA_CTL_GetFullVersionString();
```

Description:

This function returns the version of the library as a null terminated string.

Parameters:

none

Example:

```
cout << "version is: " << SA_CTL_GetFullVersionString() << endl;
```

3.2.2 SA_CTL_GetResultInfo

Interface:

```
const char* SA_CTL_GetResultInfo(
      SA_CTL_Result_t result
);
```

Description:

All functions of the library return a result code that indicates success or failure of execution. This function may be used to translate a result code into a human readable text string, e.g. to be output on a console or a GUI element.

Parameters:

- *result* (SA_CTL_Result_t), **input**: The error code.

Example:

```
SA_CTL_Result_t result;
SA_CTL_DeviceHandle_t dHandle;
result = SA_CTL_Open(&dHandle, "usb:sn:MCS2-00000001", "");
if (result != SA_CTL_ERROR_NONE) {
  cout << "Error occurred: " << SA_CTL_GetResultInfo(result) << endl;
}
```

3.2.3 SA_CTL_GetEventInfo

Interface:

```
const char* SA_CTL_GetEventInfo(
      const SA_CTL_Event_t *event
);
```

Description:

On successful return of a call to `SA_CTL_WaitForEvent` this function may be used to translate an event into a human readable text string, e.g. to be output on a console or a GUI element.

NOTICE
The string returned by this function resides in thread-local storage and remains valid only until the next call of this function.

Parameters:

- *event* (const SA_CTL_Event_t *), **input**: Pointer to a buffer which holds an event returned from `SA_CTL_WaitForEvent`

Example:

```
SA_CTL_Event_t event;
SA_CTL_Result_t result = SA_CTL_WaitForEvent(
  dHandle,
  &event,
  SA_CTL_INFINITE
);
if (result == SA_CTL_ERROR_NONE) {
  cout << "Received Event: " << SA_CTL_GetEventInfo(&event);
  cout << endl;
}
```

See also:

`SA_CTL_WaitForEvent`

3.2.4 SA_CTL_FindDevices

Interface:

```
SA_CTL_Result_t SA_CTL_FindDevices(
      const char *options,
      char *deviceList,
      size_t *deviceListLen
);
```

Description:

This function writes a list of locator strings of devices that are connected to the PC into *deviceList*. The function lists devices with a USB or ethernet interface. The *options* parameter contains a list of configuration options for the find procedure. The caller must pass a pointer to a `char` buffer in *deviceList* and set *deviceListLen* to the size of the buffer. On success the function writes a list of device locators into *deviceList* and the number of characters written into *deviceListLen*. If the supplied buffer is too small to contain the generated list, the buffer will contain no valid content but *deviceListLen* contains the required buffer size (in characters).

NOTICE
For devices with ethernet interface the Network Discover Mode must be set to passive or active mode to enable the find procedure.

Parameters:

- *options* (const char *), **input**: Options for the find procedure. **Currently unused.**
- *deviceList* (char *), **output**: Pointer to a buffer which holds the device locators after the function has returned. The locator strings are separated by a newline character.
- *deviceListLen* (size_t *), **input/output**: Specifies the size (in bytes) of *outList* before the function call. After the function call it holds the number of characters written to *deviceList*.

Example:

```
char buffer[4096];
size_t bufferSize = sizeof(buffer);
SA_CTL_Result_t result = SA_CTL_FindDevices("",buffer,&bufferSize);
if (result == SA_CTL_ERROR_NONE) {
  // buffer holds the locator strings, separated by '\n'
  // bufferSize holds the number of characters written to the buffer
}
```

See also:

4.2.7 Network Discover Mode

3.2.5 SA_CTL_Open

Interface:

```
SA_CTL_Result_t SA_CTL_Open(
      SA_CTL_DeviceHandle_t *dHandle,
      const char *locator,
      const char *config
);
```

Description:

Establishes a connection to a device for communication. Note that the overall device state is not changed. For example, settings made in previous sessions are preserved. Even ongoing movements are not interrupted by connecting to or disconnecting from the device.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t *), **output**: Handle to the device. Must be passed to following function calls.
- *locator* (const char *), **input**: Specifies the device (see section 2.1.1).
- *config* (const char *), **input**: Currently unused.

Example:

```
SA_CTL_Result_t result;
SA_CTL_DeviceHandle_t dHandle;
result = SA_CTL_Open(&dHandle, "usb:sn:MCS2-00000001", "");
if (result == SA_CTL_ERROR_NONE) {
  // success
}
```

See also:

SA_CTL_Close

3.2.6 SA_CTL_Close

Interface:

```
SA_CTL_Result_t SA_CTL_Close(
      SA_CTL_DeviceHandle_t dHandle
);
```

Description:

Closes a previously established connection to a device.

It is safe to call this function while other threads are still using the device, e.g., waiting for an event with `SA_CTL_WaitForEvent`. All blocking functions will be unblocked and will return with an `SA_CTL_ERROR_CANCELED` error.

After calling this function the device handle becomes invalid.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.

Example:

```
SA_CTL_Result_t result;
SA_CTL_DeviceHandle_t dHandle;
result = SA_CTL_Open(&dHandle, "usb:sn:MCS2-00000001", "");
if (result == SA_CTL_ERROR_NONE) {
  // success
  result = SA_CTL_Close(dHandle);
}
```

See also:

`SA_CTL_Open`

3.2.7 SA_CTL_Cancel

Interface:

```
SA_CTL_Result_t SA_CTL_Cancel(
      SA_CTL_DeviceHandle_t dHandle
);
```

Description:

This function unblocks a waiting `SA_CTL_WaitForEvent` call. If no thread is currently waiting, the next call to `SA_CTL_WaitForEvent` will be canceled. The unblocked function will return with an `SA_CTL_ERROR_CANCELED` error.

Calling this function before `SA_CTL_Close` is not required for proper cleanup.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.

See also:

`SA_CTL_WaitForEvent`, `SA_CTL_Close`

3.2.8 SA_CTL_GetProperty_i32

Interface:

```
SA_CTL_Result_t SA_CTL_GetProperty_i32(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int32_t *value,
      size_t *ioArraySize
);
```

Description:

This function retrieves a 32-bit integer property value (array) from the device. The caller must supply a pointer to a buffer where the result should be written to as well as a size information which indicates how many values may be written into the buffer. The function then writes the resulting value(s) into the buffer and sets the size information to the number of values written.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *value* (int32_t *), **output**: Pointer to a buffer where the result should be written to.
- *ioArraySize* (size_t *), **input/output**: Pointer to a size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called. On function return it contains the number of values written to the buffer. A null pointer is allowed which implicitly indicates an array size of 1.

Example:

```
// get single value (number of bus modules)
int32_t numModules;
SA_CTL_Result_t result;
result = SA_CTL_GetProperty_i32(
  dHandle, 0, SA_CTL_PKEY_NUMBER_OF_BUS_MODULES, &numModules, 0
);
if (result == SA_CTL_ERROR_NONE) {
  // numModules holds the number of modules
}
// get value array
// firmware version properties are arrays of four int32 values
```

```
int32_t fwVersion[4];
size_t ioArraySize = 4;
result = SA_CTL_GetProperty_i32(
  dHandle,0,SA_CTL_PKEY_FIRMWARE_VERSION,fwVersion,&ioArraySize
);
if (result == SA_CTL_ERROR_NONE) {
  // ioArraySize holds the number of elements
  // fwVersion holds the firmware version (rev., update, minor, major)
}
```

See also:

SA_CTL_SetProperty_i32, SA_CTL_GetProperty_i64,
SA_CTL_GetProperty_s

3.2.9 SA_CTL_SetProperty_i32

Interface:

```
SA_CTL_Result_t SA_CTL_SetProperty_i32(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int32_t value
);
```

Description:

This function writes a 32-bit integer property value to the device.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *value* (int32_t), **input**: Value that should be written.

Example:

```
// set move mode
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_i32(
  dHandle,0,SA_CTL_PKEY_MOVE_MODE,SA_CTL_MOVE_MODE_STEP
);
if (result == SA_CTL_ERROR_NONE) {
  // move mode for channel 0 is set to step mode (open-loop)
}
```

See also:

SA_CTL_GetProperty_i32, SA_CTL_SetProperty_i64,
SA_CTL_SetProperty_s

3.2.10 SA_CTL_SetPropertyArray_i32

Interface:

```
SA_CTL_Result_t SA_CTL_SetPropertyArray_i32(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      const int32_t *values
      size_t arraySize
);
```

Description:

This function writes multiple 32-bit integer values to the device and is used for setting array type properties. The caller must supply a pointer to a buffer containing the values as well as a size information which indicates how many values reside in the buffer.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *values* (const int32_t *), **input**: Pointer to a buffer that must contain the values to be written.
- *arraySize* (size_t), **input**: Size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called.

See also:

SA_CTL_GetProperty_i32, SA_CTL_SetPropertyArray_i64

3.2.11 SA_CTL_GetProperty_i64

Interface:

```
SA_CTL_Result_t SA_CTL_GetProperty_i64(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int64_t *value,
      size_t *ioArraySize
);
```

Description:

This function retrieves a 64-bit integer property value (array) from the device. The caller must supply a pointer to a buffer where the result should be written to as well as a size information which indicates how many values may be written into the buffer. The function then writes the resulting value(s) into the buffer and sets the size information to the number of values written.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *value* (int64_t *), **output**: Pointer to a buffer where the result should be written to.
- *ioArraySize* (size_t *), **input/output**: Pointer to a size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called. On function return it contains the number of values written to the buffer. A null pointer is allowed which implicitly indicates an array size of 1.

Example:

See example on page 86.

See also:

SA_CTL_SetProperty_i64, SA_CTL_GetProperty_i32, SA_CTL_GetProperty_s

3.2.12 SA_CTL_SetProperty_i64

Interface:

```
SA_CTL_Result_t SA_CTL_SetProperty_i64(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int64_t value
);
```

Description:

This function writes a 64-bit integer property value to the device.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *value* (int64_t), **input**: Value that should be written.

Example:

See example on page 88.

See also:

SA_CTL_GetProperty_i64, SA_CTL_SetProperty_i32, SA_CTL_SetProperty_s

3.2.13 SA_CTL_SetPropertyArray_i64

Interface:

```
SA_CTL_Result_t SA_CTL_SetPropertyArray_i64(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      const int64_t *values
      size_t arraySize
);
```

Description:

This function writes multiple 64-bit integer values to the device and is used for setting array type properties. The caller must supply a pointer to a buffer containing the values as well as a size information which indicates how many values reside in the buffer.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *values* (const int64_t *), **input**: Pointer to a buffer that must contain the values to be written.
- *arraySize* (size_t), **input**: Size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called.

See also:

SA_CTL_GetProperty_i64, SA_CTL_SetPropertyArray_i32

3.2.14 SA_CTL_GetProperty_s

Interface:

```
SA_CTL_Result_t SA_CTL_GetProperty_s(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      char *value,
      size_t *ioArraySize
);
```

Description:

This function retrieves a string property value (array) from the device. The caller must supply a pointer to a buffer where the result should be written to as well as a size information which indicates how many bytes may be written into the buffer. The function then writes the resulting string(s) into the buffer and sets the size information to the number of characters written. The null termination of a string implicitly serves as a separator in case multiple strings are returned.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *value* (char *), **output**: Pointer to a buffer where the result should be written to.
- *ioArraySize* (size_t *), **input/output**: Pointer to a size value that must contain size of the value buffer (in bytes) when the function is called. On function return it contains the number of characters written to the buffer.

Example:

```
char deviceSerial[128];
size_t len = sizeof(deviceSerial);
SA_CTL_Result_t result;
result = SA_CTL_GetProperty_s(
  dHandle,0,SA_CTL_PKEY_DEVICE_SERIAL_NUMBER,deviceSerial,&len
);
if (result == SA_CTL_ERROR_NONE) {
  // deviceSerial holds the unique serial number of the device
  // len holds the length of the string
}
```

See also:

SA_CTL_SetProperty_s, SA_CTL_GetProperty_i32,
SA_CTL_GetProperty_i64

3.2.15 SA_CTL_SetProperty_s

Interface:

```
SA_CTL_Result_t SA_CTL_SetProperty_i32(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      const char *value
);
```

Description:

This function writes a string property value to the device. Note that the length of strings may never exceed 63 characters (plus a null terminator).

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed device, module or channel (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key that identifies the property.
- *value* (const char *), **input**: String that should be written.

Example:

```
SA_CTL_Result_t result;
result = SA_CTL_SetProperty_s(
  dHandle,0,SA_CTL_PKEY_DEVICE_NAME,"MyFavoriteController"
);
if (result == SA_CTL_ERROR_NONE) {
  // success
}
```

See also:

SA_CTL_GetProperty_s, SA_CTL_SetProperty_i32, SA_CTL_SetProperty_i64

3.2.16 SA_CTL_RequestReadProperty

Interface:

```
SA_CTL_Result_t SA_CTL_RequestReadProperty(
       SA_CTL_DeviceHandle_t dHandle,
       int8_t idx,
       SA_CTL_PropertyKey_t pkey,
       SA_CTL_RequestID_t *rID,
       SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function requests to read a property value (array) from the device and can be used for asynchronous (non-blocking) access. The caller must supply a pointer to a buffer where the request ID should be written to. Received values can be accessed later via the obtained request ID and the corresponding `SA_CTL_ReadProperty_x` functions.

The advantage of this method is that the application may request several property values in fast succession and then perform other tasks before blocking on the reception of the results.

NOTICE

The correct `SA_CTL_ReadProperty_x` function must be used depending on the data type of the requested property. Otherwise the read will fail with a `SA_CTL_ERROR_INVALID_DATA_TYPE` error.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel or module (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key of the property that is requested.
- *rID* (SA_CTL_RequestID_t *), **output**: Pointer to a request ID.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Optional ID to a transmit buffer. If unused set to zero.

Example:

```
// Note: to keep the example clear, we omit processing the result codes
SA_CTL_Request_t rID[2];
int64_t position;
int32_t state;
// Issue requests for the two properties "position" and "channel state"
SA_CTL_RequestReadProperty(
  dHandle, 0, SA_CTL_PKEY_POSITION, &rID[0], 0
);
SA_CTL_RequestReadProperty(
  dHandle, 0, SA_CTL_PKEY_CHANNEL_STATE, &rID[1], 0
);
// process other tasks
// ...
// Receive the results
SA_CTL_ReadProperty_i64(dHandle, rID[0], &position, 0);
SA_CTL_ReadProperty_i32(dHandle, rID[1], &state, 0);
```

See also:

SA_CTL_ReadProperty_i32, SA_CTL_ReadProperty_i64,
SA_CTL_ReadProperty_s

3.2.17 SA_CTL_ReadProperty_i32

Interface:

```
SA_CTL_Result_t SA_CTL_ReadProperty_i32(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_RequestID_t rID,
      int32_t *value,
      size_t *ioArraySize
);
```

Description:

This function reads a 32-bit integer property value (array) that has previously been requested using `SA_CTL_RequestReadProperty`.

NOTICE
While the request-function is non-blocking the read functions block until the desired data has arrived.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *rID* (SA_CTL_RequestID_t), **input**: ID of the addressed request.
- *value* (int32_t *), **output**: Pointer to a buffer where the result should be written to.
- *ioArraySize* (size_t *), **input/output**: Pointer to a size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called. On function return it contains the number of values written to the buffer. A null pointer is allowed which implicitly indicates an array size of 1.

Example:

See example on page 97.

See also:

`SA_CTL_RequestReadProperty`, `SA_CTL_ReadProperty_i64`, `SA_CTL_ReadProperty_s`

3.2.18 SA_CTL_ReadProperty_i64

Interface:

```
SA_CTL_Result_t SA_CTL_ReadProperty_i64(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_RequestID_t rID,
      int64_t *value,
      size_t *ioArraySize
);
```

Description:

This function reads a 64-bit integer property value (array) that has previously been requested using `SA_CTL_RequestReadProperty`.

NOTICE
While the request-function is non-blocking the read functions block until the desired data has arrived.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *rID* (SA_CTL_RequestID_t), **input**: ID of the addressed request.
- *value* (int64_t *), **output**: Pointer to a buffer where the result should be written to.
- *ioArraySize* (size_t *), **input/output**: Pointer to a size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called. On function return it contains the number of values written to the buffer. A null pointer is allowed which implicitly indicates an array size of 1.

Example:

See example on page 97.

See also:

`SA_CTL_RequestReadProperty`, `SA_CTL_ReadProperty_i32`, `SA_CTL_ReadProperty_s`

3.2.19 SA_CTL_ReadProperty_s

Interface:

```
SA_CTL_Result_t SA_CTL_ReadProperty_s(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_RequestID_t rID,
      char *value,
      size_t *ioStringSize
);
```

Description:

This function reads a string property value (array) that has previously been requested using `SA_CTL_RequestReadProperty`.

NOTICE
While the request-function is non-blocking the read functions block until the desired data has arrived.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *rID* (SA_CTL_RequestID_t), **input**: ID of the addressed request.
- *value* (char *), **output**: Pointer to a buffer where the result should be written to.
- *ioStringSize* (size_t *), **input/output**: Pointer to a size value that must contain size of the value buffer (in bytes) when the function is called. On function return it contains the number of characters written to the buffer.

Example:

```
// Note: to keep the example simple, we omit processing the result codes
SA_CTL_Request_t rID;
char deviceSerial[128];
size_t len = sizeof(deviceSerial);
// Issue request for the "device serial number" property
SA_CTL_RequestReadProperty(
  dHandle, 0, SA_CTL_PKEY_DEVICE_SERIAL_NUMBER, &rID, 0
);
// process other tasks
// ...
```

```
// Receive the result
SA_CTL_ReadProperty_s(dHandle, rID, deviceSerial, &len);
```

See also:

SA_CTL_RequestReadProperty, SA_CTL_ReadProperty_i32, SA_CTL_ReadProperty_i64

3.2.20 SA_CTL_RequestWriteProperty_i32

Interface:

```
SA_CTL_Result_t SA_CTL_RequestWriteProperty_i32(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int32_t value,
      SA_CTL_RequestID_t *rID,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function writes a 32-bit integer value to the device and can be used for asynchronous (non-blocking) access. The caller can supply a pointer to a buffer where the request ID should be written to. The result (whether the write was successful or not) can be accessed later by passing the obtained request ID to the `SA_CTL_WaitForWrite` function.

The advantage of this method is that the application may write several property values in fast succession and then perform other tasks before blocking on the reception of the results.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel or module (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key of the property that should be accessed.
- *value* (int32_t), **input**: Value that should be written.
- *rID* (SA_CTL_RequestID_t *), **output**: Pointer to a request ID. Can be null pointer for call-and-forget mechanism (see section 2.3.4).
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Optional ID to a transmit buffer. If unused set to zero.

Example:

```
SA_CTL_Result_t result;
SA_CTL_RequestID_t rID;
int8_t channel;
int64_t holdTime = 5000;
// Request to set hold time to 5 seconds
result = SA_CTL_RequestWriteProperty_i32(
  dHandle, channel, SA_CTL_PKEY_HOLD_TIME, holdTime, &rID, 0
```

```
);
// process other tasks
// ...
// Wait for the result to arrive
result = SA_CTL_WaitForWrite(dHandle, rID);
```

See also:

SA_CTL_WaitForWrite, SA_CTL_RequestWriteProperty_i64, SA_CTL_RequestWriteProperty_s

3.2.21 SA_CTL_RequestWriteProperty_i64

Interface:

```
SA_CTL_Result_t SA_CTL_RequestWriteProperty_i64(
       SA_CTL_DeviceHandle_t dHandle,
       int8_t idx,
       SA_CTL_PropertyKey_t pkey,
       int64_t value,
       SA_CTL_RequestID_t *rID,
       SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function writes a 64-bit integer value to the device and can be used for asynchronous (non-blocking) access. The caller can supply a pointer to a buffer where the request ID should be written to. The result (whether the write was successful or not) can be accessed later by passing the obtained request ID to the `SA_CTL_WaitForWrite` function.

The advantage of this method is that the application may write several property values in fast succession and then perform other tasks before blocking on the reception of the results.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel or module (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key of the property that should be accessed.
- *value* (int64_t), **input**: Value that should be written.
- *rID* (SA_CTL_RequestID_t *), **output**: Pointer to a request ID. Can be null pointer for call-and-forget mechanism (see section 2.3.4).
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Optional ID to a transmit buffer. If unused set to zero.

Example:

See example on page 102.

See also:

`SA_CTL_WaitForWrite`, `SA_CTL_RequestWriteProperty_i32`, `SA_CTL_RequestWriteProperty_s`

3.2.22 SA_CTL_RequestWriteProperty_s

Interface:

```
SA_CTL_Result_t SA_CTL_RequestWriteProperty_s(
       SA_CTL_DeviceHandle_t dHandle,
       int8_t idx,
       SA_CTL_PropertyKey_t pkey,
       const char *value,
       SA_CTL_RequestID_t *rID,
       SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function writes a string value to the device and can be used for asynchronous (non-blocking) access. The caller can supply a pointer to a buffer where the request ID should be written to. The result (whether the write was successful or not) can be accessed later by passing the obtained request ID to the `SA_CTL_WaitForWrite` function.

The advantage of this method is that the application may write several property values in fast succession and then perform other tasks before blocking on the reception of the results.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel or module (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key of the property that should be accessed.
- *value* (const char *), **input**: Value that should be written.
- *rID* (SA_CTL_RequestID_t *), **output**: Pointer to a request ID. Can be null pointer for call-and-forget mechanism (see section 2.3.4).
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Optional ID to a transmit buffer. If unused set to zero.

Example:

See example on page 102.

See also:

`SA_CTL_WaitForWrite`, `SA_CTL_RequestWriteProperty_i32`, `SA_CTL_RequestWriteProperty_i64`

3.2.23 SA_CTL_RequestWritePropertyArray_i32

Interface:

```
SA_CTL_Result_t SA_CTL_RequestWritePropertyArray_i32(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int32_t *values,
      size_t arraySize,
      SA_CTL_RequestID_t *rID,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function writes multiple 32-bit integer values to the device and can be used for asynchronous (non-blocking) access of array type properties. The caller must supply a pointer to a buffer containing the values as well as a size information which indicates how many values reside in the buffer. Furthermore a pointer to a buffer where the request ID should be written to can be provided. The result (whether the write was successful or not) can be accessed later by passing the obtained request ID to the `SA_CTL_WaitForWrite` function.

The advantage of this method is that the application may write several property values in fast succession and then perform other tasks before blocking on the reception of the results.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel or module (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key of the property that should be accessed.
- *values* (int32_t *), **input**: Pointer to a buffer that must contain the values to be written.
- *arraySize* (size_t), **input**: Size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called.
- *rID* (SA_CTL_RequestID_t *), **output**: Pointer to a request ID. Can be null pointer for call-and-forget mechanism (see section 2.3.4).
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Optional ID to a transmit buffer. If unused set to zero.

See also:

`SA_CTL_WaitForWrite`, `SA_CTL_RequestWritePropertyArray_i64`

3.2.24 SA_CTL_RequestWritePropertyArray_i64

Interface:

```
SA_CTL_Result_t SA_CTL_RequestWritePropertyArray_i64(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_PropertyKey_t pkey,
      int64_t *values,
      size_t arraySize,
      SA_CTL_RequestID_t *rID,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function writes multiple 64-bit integer values to the device and can be used for asynchronous (non-blocking) access of array type properties. The caller must supply a pointer to a buffer containing the values as well as a size information which indicates how many values reside in the buffer. Furthermore a pointer to a buffer where the request ID should be written to can be provided. The result (whether the write was successful or not) can be accessed later by passing the obtained request ID to the `SA_CTL_WaitForWrite` function.

The advantage of this method is that the application may write several property values in fast succession and then perform other tasks before blocking on the reception of the results.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel or module (see section 2.2).
- *pkey* (SA_CTL_PropertyKey_t), **input**: Key of the property that should be accessed.
- *values* (int64_t *), **input**: Pointer to a buffer that must contain the values to be written.
- *arraySize* (size_t), **input**: Size value that must contain the size of the value buffer (in number of elements, not number of bytes) when the function is called.
- *rID* (SA_CTL_RequestID_t *), **output**: Pointer to a request ID. Can be null pointer for call-and-forget mechanism (see section 2.3.4).
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Optional ID to a transmit buffer. If unused set to zero.

See also:

`SA_CTL_WaitForWrite`, `SA_CTL_RequestWritePropertyArray_i32`

3.2.25 SA_CTL_WaitForWrite

Interface:

```
SA_CTL_Result_t SA_CTL_WaitForWrite(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_RequestID_t rID
);
```

Description:

This function returns the result of a property write access that has previously been requested using the data type specific `SA_CTL_RequestWriteProperty_x` function.

NOTICE
While the request function is non-blocking the `SA_CTL_WaitForWrite` function blocks until the desired result has arrived.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *rID* (SA_CTL_RequestID_t), **input**: ID of the addressed request.

Example:

See example on page 102.

See also:

`SA_CTL_RequestWriteProperty_i32`, `SA_CTL_RequestWriteProperty_i64`, `SA_CTL_RequestWriteProperty_s`

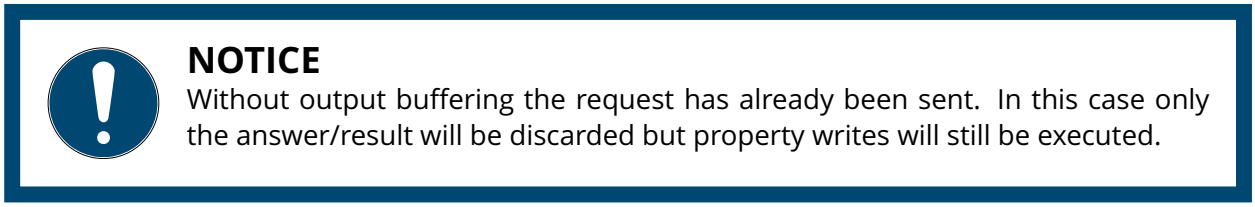
3.2.26 SA_CTL_CancelRequest

Interface:

```
SA_CTL_Result_t SA_CTL_CancelRequest(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_RequestID_t rID
);
```

Description:

This function cancels a non-blocking read or write request.



Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *rID* (SA_CTL_RequestID_t), **input**: ID of the addressed request.

Example:

```
SA_CTL_Result_t result;
SA_CTL_RequestID_t rID;
// Request to set hold time to 5 seconds
result = SA_CTL_RequestWriteProperty_i32(
  dHandle, 0, SA_CTL_PKEY_HOLD_TIME, 5000, &rID, 0
);
// process other tasks
// ...
// We are not interested in the result anymore and discard the request
result = SA_CTL_CancelRequest(dHandle, rID);
```

See also:

SA_CTL_RequestWriteProperty_i32, SA_CTL_RequestWriteProperty_i64, SA_CTL_RequestWriteProperty_s, SA_CTL_RequestReadProperty

3.2.27 SA_CTL_CreateOutputBuffer

Interface:

```
SA_CTL_Result_t SA_CTL_CreateOutputBuffer(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_TransmitHandle_t *tHandle
);
```

Description:

Creates an output buffer for optimizing communication throughput with the device using the asynchronous command set. After creation the retrieved transmit handle can be used to choose whether a command is to be buffered or sent directly. A buffered command is not sent to the device immediately. Instead, the data is held back and stored in the internal buffer. You may accumulate several commands and then call `SA_CTL_FlushOutputBuffer` to initiate the transmission or `SA_CTL_CancelOutputBuffer` to cancel the output buffer.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *tHandle* (SA_CTL_TransmitHandle_t *), **output**: Pointer to a transmit handle.

Example:

```
// Note: to keep the example simple, we omit processing the result codes
SA_CTL_TransmitHandle_t tHandle;
SA_CTL_CreateOutputBuffer(dHandle, &tHandle);
SA_CTL_Move(dHandle, 0, 1000000, tHandle);
SA_CTL_Move(dHandle, 1, -1000000, tHandle);
// move commands have not been transmitted yet.
SA_CTL_FlushOutputBuffer(dHandle, tHandle);
// move commands have been transmitted and will be executed.
```

See also:

`SA_CTL_FlushOutputBuffer`, `SA_CTL_CancelOutputBuffer`

3.2.28 SA_CTL_FlushOutputBuffer

Interface:

```
SA_CTL_Result_t SA_CTL_FlushOutputBuffer(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

Initiates the transmission of all commands stored in the output buffer that is associated with the given transmit handle.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer.

Example:

```
SA_CTL_Result_t result;
SA_CTL_TransmitHandle_t tHandle;
result = SA_CTL_CreateOutputBuffer(dHandle, &tHandle);
if (result == SA_CTL_ERROR_NONE) {
  // tHandle now holds a valid transmit handle
}
// append commands to buffer here
result = SA_CTL_FlushBuffer(dHandle, tHandle);
if (result == SA_CTL_ERROR_NONE) {
  // buffer is now flushed and the transmit handle released
}
// process generated answers/events
```

See also:

SA_CTL_CreateOutputBuffer, SA_CTL_CancelOutputBuffer

3.2.29 SA_CTL_CancelOutputBuffer

Interface:

```
SA_CTL_Result_t SA_CTL_CancelOutputBuffer(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

Discards all buffered commands and releases the associated transmit handle.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer.

Example:

```
SA_CTL_Result_t result;
SA_CTL_TransmitHandle_t tHandle;
result = SA_CTL_CreateOutputBuffer(dHandle, &tHandle);
if (result == SA_CTL_ERROR_NONE) {
  // tHandle now holds a valid transmit handle
}
// append commands to buffer here
result = SA_CTL_CancelBuffer(dHandle, tHandle);
if (result == SA_CTL_ERROR_NONE) {
  // all buffered commands are discarded and the transmit handle released
}
```

See also:

SA_CTL_CreateOutputBuffer, SA_CTL_FlushOutputBuffer

3.2.30 SA_CTL_OpenCommandGroup

Interface:

```
SA_CTL_Result_t SA_CTL_OpenCommandGroup(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_TransmitHandle_t *tHandle,
      uint32_t triggerMode
);
```

Description:

Opens a command group that can be used to combine multiple asynchronous commands into an atomic group. A trigger mode can be set to select between different modes to start the groups execution. After creation the retrieved transmit handle can be used to choose whether a command is to be grouped or sent directly. You may accumulate several commands and then call `SA_CTL_CloseCommandGroup` to activate or `SA_CTL_CancelCommandGroup` to cancel the command group.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *tHandle* (SA_CTL_TransmitHandle_t *), **output**: Pointer to a transmit handle.
- *triggerMode* (uint32_t), **input**: Desired trigger mode for this command group. Must be either `SA_CTL_CMD_GROUP_TRIGGER_MODE_DIRECT` (`0`) or `SA_CTL_CMD_GROUP_TRIGGER_MODE_EXTERNAL` (`1`).

Example:

```
SA_CTL_Result_t result;
SA_CTL_TransmitHandle_t tHandle;
result = SA_CTL_OpenCommandGroup(
  dHandle,&tHandle,SA_CTL_CMD_GROUP_TRIGGER_MODE_DIRECT
);
if (result == SA_CTL_ERROR_NONE) {
  // tHandle now holds a valid transmit handle
}
```

See also:

`SA_CTL_CloseCommandGroup`, `SA_CTL_CancelCommandGroup`

3.2.31 SA_CTL_CloseCommandGroup

Interface:

```
SA_CTL_Result_t SA_CTL_CloseCommandGroup(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

Closes and eventually executes the assembled command group depending on the configured trigger mode.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer.

Example:

```
SA_CTL_Result_t result;
SA_CTL_TransmitHandle_t tHandle;
result = SA_CTL_OpenCommandGroup(
  dHandle,&tHandle,SA_CTL_CMD_GROUP_TRIGGER_MODE_DIRECT
);
if (result == SA_CTL_ERROR_NONE) {
  // tHandle now holds a valid transmit handle
}
// append commands to buffer here
result = SA_CTL_CloseCommandGroup(dHandle, tHandle);
if (result == SA_CTL_ERROR_NONE) {
  // command group is now activated. since the command group is
  // triggered directly, it is executed right away.
}
// process other tasks
// ...
// optional: wait for the SA_CTL_EVENT_CMD_GROUP_TRIGGERED event
// process answers/events to commands
```

See also:

SA_CTL_OpenCommandGroup, SA_CTL_CancelCommandGroup

3.2.32 SA_CTL_CancelCommandGroup

Interface:

```
SA_CTL_Result_t SA_CTL_CancelCommandGroup(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

Discards all buffered commands and releases the associated transmit handle.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer.

Example:

```
SA_CTL_Result_t result;
SA_CTL_TransmitHandle_t tHandle;
result = SA_CTL_OpenCommandGroup(
  dHandle,&tHandle,SA_CTL_CMD_GROUP_TRIGGER_MODE_DIRECT
);
if (result == SA_CTL_ERROR_NONE) {
  // tHandle now holds a valid transmit handle
}
// append commands to buffer here
result = SA_CTL_CancelCommandGroup(dHandle, tHandle);
if (result == SA_CTL_ERROR_NONE) {
  // all buffered commands are discarded and the transmit handle released
}
```

See also:

SA_CTL_OpenCommandGroup, SA_CTL_CloseCommandGroup

3.2.33 SA_CTL_WaitForEvent

Interface:

```
SA_CTL_Result_t SA_CTL_WaitForEvent(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_Event_t *event,
      uint32_t timeout
);
```

Description:

This function blocks until the device reports an event. Usually this function is used in a separate thread. The function returns when:

- An event has occurred within the given timeout. In this case the return value of the function will be `SA_CTL_ERROR_NONE` and the output parameter *event* will hold the event that occurred. See section 2.4 "Event Notifications" for the structure of events.
- No event occurred within the given timeout. In this case the return value of the function will be `SA_CTL_ERROR_TIMEOUT` and the *event* parameter is undefined.
- The call is canceled with a call of `SA_CTL_Cancel` from another application thread. In this case the return value of the function will be `SA_CTL_ERROR_CANCELED` and the *event* parameter is undefined. This is typically useful when the application is to be terminated and the event handling thread must be unblocked for a proper cleanup.

NOTICE

This function cannot be called simultaneously using multiple threads (for the same device handle). If a second thread tries to call this function, then a `SA_CTL_ERROR_THREAD_LIMIT_REACHED` error will be returned.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *event* (SA_CTL_Event_t *), **output**: Event that occurred.
- *timeout* (uint32_t), **input**: Maximum time to wait for an event to occur. The timeout is given in milliseconds. The special value `SA_CTL_INFINITE` is also valid.

Example:

```
// thread 1:
SA_CTL_Event_t event;
SA_CTL_Result_t result;
result = SA_CTL_WaitForEvent(dHandle,&event,SA_CTL_INFINITE);
if (result == SA_CTL_ERROR_CANCELED) {
  // SA_CTL_WaitForEvent was canceled before an event occurred
}
```

```
// thread 2:
// wake up waiting thread 1
SA_CTL_Result_t result = SA_CTL_Cancel(dHandle);
```

See also:

SA_CTL_Cancel

3.2.34 SA_CTL_Calibrate

Interface:

```
SA_CTL_Result_t SA_CTL_Calibrate(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This movement function performs a calibration routine for a channel. Before calling this function the calibration options should be configured. See section 2.6.1 "Calibrating" for more information.

NOTICE
The function call returns immediately, without waiting for the movement to complete. The calibration may however take a few seconds to complete. Therefore the `SA_CTL_CH_STATE_BIT_CALIBRATING` in the Channel State can be monitored to determine the end of the calibration sequence.

CAUTION
As a safety precaution, make sure that the positioner has enough freedom to move without damaging other equipment.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer. If unused set to zero.

Example:

```
SA_CTL_Result_t result;
// Set calibration mode for channel 0 (start direction: forward)
result = SA_CTL_SetProperty_i32(
  dHandle, 0, SA_CTL_PKEY_CALIBRATION_OPTIONS, 0
);
```

```
if (result == SA_CTL_ERROR_NONE) {
  // calibration mode is now set
}
// Start calibration sequence
result = SA_CTL_Calibrate(dHandle, 0, 0);
if (result == SA_CTL_ERROR_NONE) {
  // calibration is now started (function call returns immediately)
}
```

3.2.35 SA_CTL_Reference

Interface:

```
SA_CTL_Result_t SA_CTL_Reference(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This movement function may be used to move the positioner to a known physical position. Before calling this function the reference options should be configured. See section 2.6.2 "Referencing" for more information.

NOTICE

The function call returns immediately, without waiting for the movement to complete. The `SA_CTL_CH_STATE_BIT_REFERENCING` in the Channel State can be monitored to determine the end of the referencing sequence. If the command was successful the `SA_CTL_CH_STATE_BIT_IS_REFERENCED` in the Channel State will be set. This bit can also be checked to determine whether it is necessary to perform the referencing sequence.

CAUTION

As a safety precaution, make sure that the positioner has enough freedom to move without damaging other equipment.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer. If unused set to zero.

Example:

```
SA_CTL_Result_t result;
// Set find reference mode for channel 0 (default is 0)
result = SA_CTL_SetProperty_i32(
  dHandle, 0, SA_CTL_PKEY_REFERENCING_OPTIONS, 0
);
if (result == SA_CTL_ERROR_NONE) {
  // desired reference mode is now set
}
// Start referencing sequence
result = SA_CTL_Reference(dHandle, 0, 0);
if (result == SA_CTL_ERROR_NONE) {
  // referencing sequence has started (function call returns immediately)
}
```

3.2.36 SA_CTL_Move

Interface:

```
SA_CTL_Result_t SA_CTL_Move(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      int64_t moveValue,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function instructs a positioner to move according to the current move configuration. The move mode as well as corresponding parameters (e.g. Frequency, Velocity, HoldTime, etc.) have to be configured beforehand using the `SA_CTL_SetProperty_x` functions. See section 2.6 "Moving Positioners" for more information.

NOTICE

The function call returns immediately, without waiting for the movement to complete. The Channel State bits `SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING` and `SA_CTL_CH_STATE_BIT_CLOSED_LOOP_ACTIVE` can be monitored to determine the end of the movement.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel.
- *moveValue* (int64_t), **input**: Interpretation depends on the configured move mode.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer. If unused set to zero.

Example:

```
// Note: to keep the example simple, we omit processing the result codes
// Set move mode
SA_CTL_SetProperty_i32(
  dHandle, 0, SA_CTL_PKEY_MOVE_MODE, SA_CTL_MOVE_MODE_CL_RELATIVE
);
// Set move velocity [in pm/s]
SA_CTL_SetProperty_i64(
```

```
  dHandle, 0, SA_CTL_PKEY_MOVE_VELOCITY, 500000000
);
// Set move acceleration [in pm/s2],
// a value of 0 disables the acceleration control
SA_CTL_SetProperty_i64(
  dHandle, 0, SA_CTL_PKEY_MOVE_ACCELERATION, 0
);
// Start actual movement, moveValue holds relative position (in pm)
SA_CTL_Move(dhandle, 0, 500000000, 0);
```

3.2.37 SA_CTL_Stop

Interface:

```
SA_CTL_Result_t SA_CTL_Stop(
      SA_CTL_DeviceHandle_t dHandle,
      int8_t idx,
      SA_CTL_TransmitHandle_t tHandle
);
```

Description:

This function stops any ongoing movement of a positioner. It also stops the hold position feature of a closed-loop command.

Note for closed-loop movements with acceleration control enabled: The first `stop` command sent while moving triggers the positioner to come to a halt by decelerating to zero. A second `stop` command triggers a hard stop (*emergency stop*).

NOTICE
The function call returns immediately, without waiting for the stop to complete. The `SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING` in the Channel State can be monitored to determine the end of the movement.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *idx* (int8_t), **input**: Index of the addressed channel.
- *tHandle* (SA_CTL_TransmitHandle_t), **input**: Handle of the addressed transmit buffer. If unused set to zero.

Example:

```
int8_t channel = 0;
SA_CTL_Result_t result;
result = SA_CTL_Stop(dHandle, channel, 0);
if (result == SA_CTL_ERROR_NONE) {
  // stop command is now being executed
}
```

3.2.38 SA_CTL_OpenStream

Interface:

```
SA_CTL_Result_t SA_CTL_OpenStream(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_StreamHandle_t *sHandle,
      uint32_t triggerMode
);
```

Description:

This function opens a stream to the device. It is used for trajectory streaming (see section 2.15). The caller must supply a pointer to a buffer where the stream handle should be written to. A trigger mode can be set to select between different modes to start and synchronize the streaming process.

NOTICE
The desired stream base rate has to be configured before calling this function.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *sHandle* (SA_CTL_StreamHandle_t *), **output**: Pointer to a stream handle.
- *triggerMode* (uint32_t), **input**: Desired trigger mode. May be one of
  `SA_CTL_STREAM_TRIGGER_MODE_DIRECT` (0),
  `SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_ONCE` (1),
  `SA_CTL_STREAM_TRIGGER_MODE_EXTERNAL_SYNC` (2).

Example:

```
SA_CTL_Result_t result;
SA_CTL_StreamHandle_t sHandle;
result = SA_CTL_OpenStream(
  dHandle,
  &sHandle,
  SA_CTL_STREAM_TRIGGER_MODE_DIRECT
);
if (result == SA_CTL_ERROR_NONE) {
  // stream is now opened
}
```

See also:

SA_CTL_StreamFrame, SA_CTL_CloseStream, SA_CTL_AbortStream

3.2.39 SA_CTL_StreamFrame

Interface:

```
SA_CTL_Result_t SA_CTL_StreamFrame(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_StreamHandle_t sHandle,
      uint8_t *frameData,
      uint32_t frameSize
);
```

Description:

This function supplies the device with stream data by sending one frame per function call. A frame contains the data for one interpolation point which must be assembled by concatenating elements of the following tuple:

- *Channel Index* (1 byte): The channel that receives the following position.
- *Position* (8 byte): A position that belongs to the current interpolation point.

See section 2.15 "Trajectory Streaming" for more information.

NOTICE
This function may block if the flow control needs to throttle the data rate. The function returns as soon as the frame was transmitted to the controller.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *sHandle* (SA_CTL_StreamHandle_t), **input**: Handle of the addressed stream.
- *frameData* (uint8_t *), **input**: Pointer to the frame data buffer.
- *frameSize* (uint32_t), **input**: Size of the given frame (in bytes).

Example:

```
SA_CTL_Result_t result;
// create frame data array for 2 channel/position tuples
uint8_t frameData[2*(1+8)];
// fill frame with data
// ...
// send frame
```

```
result = SA_CTL_StreamFrame(
  dHandle, sHandle, frameData, sizeof(frameData)
);
if (result == SA_CTL_ERROR_NONE) {
  // frame successfully sent to the device
}
```

See also:

SA_CTL_OpenStream, SA_CTL_CloseStream, SA_CTL_AbortStream

3.2.40 SA_CTL_CloseStream

Interface:

```
SA_CTL_Result_t SA_CTL_CloseStream(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_StreamHandle_t sHandle
);
```

Description:

This function closes a stream. For the device this marks the end of the stream. After having processed the remaining buffered interpolation points the stream is finished. See section 2.15 for more information.

NOTICE
If the stream is not closed properly, the device will generate a buffer underflow error after the last frame has been processed.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *sHandle* (SA_CTL_StreamHandle_t), **input**: Handle of the addressed stream.

Example:

```
SA_CTL_StreamHandle_t sHandle;
SA_CTL_Result_t result;
result = SA_CTL_OpenStream(
  dHandle,
  &sHandle,
  SA_CTL_STREAM_TRIGGER_MODE_DIRECT
);
if (result != SA_CTL_ERROR_NONE) {
  // handle error
}
// stream frames
// ...
result = SA_CTL_CloseStream(dHandle, sHandle);
// remaining interpolation points are now processed
```

See also:

SA_CTL_OpenStream, SA_CTL_StreamFrame, SA_CTL_AbortStream

3.2.41 SA_CTL_AbortStream

Interface:

```
SA_CTL_Result_t SA_CTL_AbortStream(
      SA_CTL_DeviceHandle_t dHandle,
      SA_CTL_StreamHandle_t sHandle
);
```

Description:

This function aborts a stream. Thus all movements are stopped immediately and remaining buffered interpolation points are discarded.

Parameters:

- *dHandle* (SA_CTL_DeviceHandle_t), **input**: Handle of the addressed device.
- *sHandle* (SA_CTL_StreamHandle_t), **input**: Handle of the addressed stream.

Example:

```
SA_CTL_StreamHandle_t sHandle;
SA_CTL_Result_t result;
result = SA_CTL_OpenStream(
  dHandle,
  &sHandle,
  SA_CTL_STREAM_TRIGGER_MODE_DIRECT
);
if (result != SA_CTL_ERROR_NONE) {
  // handle error
}
// stream frames
// ...
result = SA_CTL_AbortStream(dHandle, sHandle);
// stream is aborted immediately
```

See also:

SA_CTL_OpenStream, SA_CTL_StreamFrame, SA_CTL_CloseStream

4 *PROPERTY REFERENCE*

4.1 Property Summary

Table 4.1 – Property Summary

Property	Code	Type	Idx	Access	CG[1]	NV[2]	Page
Device Properties							
Number of Channels	0x020F0017	I32	Dev	R	-	-	137
Number of Bus Modules	0x020F0016	I32	Dev	R	-	-	137
Device State	0x020F000F	I32	Dev	R	-	-	138
Device Serial Number	0x020F005E	String	Dev	R	-	-	139
Device Name	0x020F003D	String	Dev	RW	-	X	140
Emergency Stop Mode	0x020F0088	I32	Dev	RW	-	-	141
Network Discover Mode	0x020F0159	I32	Dev	RW	-	X	142
Module Properties							
Power Supply Enabled	0x02030010	I32	Mod	RW	X	-	143
Module State	0x0203000F	I32	Mod	R	X	-	144
Number of Bus Module Channels	0x02030017	I32	Mod	R	X	-	145
Positioner Properties							
Amplifier Enabled	0x0302000D	I32	Ch	RW	X	-	145
Positioner Control Options	0x0302005D	I32	Ch	RW	X	X	146
Actuator Mode	0x03020019	I32	Ch	RW	-	-	147
Control Loop Input	0x03020018	I32	Ch	RW	X	X	149
Sensor Input Select	0x03020018	I32	Ch	RW	X	X	149
Positioner Type	0x0302003C	I32	Ch	RW	X	X	151
Positioner Type Name	0x0302003D	String	Ch	R	-	-	152
Move Mode	0x03050087	I32	Ch	RW	X	-	152
Channel State	0x0305000F	I32	Ch	R	X	-	154
Position	0x0305001D	I64	Ch	RW	X	-	155

Continued on next page

Table 4.1 – *Continued from previous page*

Property	Code	Type	Idx	Access	CG[1]	NV[2]	Page
Target Position	0x0305001E	I64	Ch	R	X	-	156
Scan Position	0x0305001F	I64	Ch	R	X	-	156
Scan Velocity	0x0305002A	I64	Ch	RW	X	-	157
Hold Time	0x03050028	I32	Ch	RW	X	-	158
Move Velocity	0x03050029	I64	Ch	RW	X	-	159
Move Acceleration	0x0305002B	I64	Ch	RW	X	-	160
Max Closed Loop Frequency	0x0305002F	I32	Ch	RW	X	-	161
Default Max Closed Loop Frequency	0x03050057	I32	Ch	RW	X	X	162
Step Frequency	0x0305002E	I32	Ch	RW	X	-	163
Step Amplitude	0x03050030	I32	Ch	RW	X	-	163
Following Error Limit	0x03050055	I64	Ch	RW	X	X	164
Broadcast Stop Options	0x0305005D	I32	Ch	RW	X	-	165
Sensor Power Mode	0x03080019	I32	Ch	RW	X	X	166
Sensor Power Save Delay	0x03080054	I32	Ch	RW	X	X	167
Position Mean Shift	0x03090022	I32	Ch	RW	X	X	168
Safe Direction	0x03090027	I32	Ch	RW	X	X	169
Control Loop Input Sensor Value	0x0302001D	I64	Ch	R	X	-	170
Control Loop Input Aux Value	0x030200B2	I64	Ch	R	X	-	171
Target To Zero Voltage Hold Threshold	0x030200B9	I32	Ch	RW	X	X	172
Scale Properties							
Logical Scale Offset	0x02040024	I64	Ch	RW	X	X	173
Logical Scale Inversion	0x02040025	I32	Ch	RW	X	X	174
Range Limit Min	0x02040020	I64	Ch	RW	X	-	175
Range Limit Max	0x02040021	I64	Ch	RW	X	-	175
Calibration Properties							
Calibration Options	0x0306005D	I32	Ch	RW	X	-	176
Signal Correction Options	0x0306001C	I32	Ch	RW	X	X	177
Referencing Properties							
Referencing Options	0x0307005D	I32	Ch	RW	X	-	179

Continued on next page

Table 4.1 – *Continued from previous page*

Property	Code	Type	Idx	Access	CG[1]	NV[2]	Page
Distance To Reference Mark	0x030700A2	I64	Ch	R	X	-	180
Distance Code Inverted	0x0307000E	I32	Ch	RW	X	X	180
Positioner Tuning and Customizing Properties							
Positioner Movement Type	0x0309003F	I32	Ch	R(W)	X	(X)	181
Positioner Is Custom Type	0x03090041	I32	Ch	R(W)	X	(X)	182
Positioner Base Unit	0x03090042	I32	Ch	R(W)	X	(X)	183
Positioner Base Resolution	0x03090043	I32	Ch	R(W)	X	(X)	184
Positioner Sensor Head Type	0x0309008E	I32	Ch	R(W)	X	(X)	185
Positioner Reference Type	0x03090048	I32	Ch	R(W)	X	(X)	186
Positioner P Gain	0x0309004B	I32	Ch	R(W)	X	(X)	187
Positioner I Gain	0x0309004C	I32	Ch	R(W)	X	(X)	188
Positioner D Gain	0x0309004D	I32	Ch	R(W)	X	(X)	189
Positioner PID Shift	0x0309004E	I32	Ch	R(W)	X	(X)	190
Positioner Anti Windup	0x0309004F	I32	Ch	R(W)	X	(X)	191
Positioner ESD Distance Threshold	0x03090050	I32	Ch	R(W)	X	(X)	192
Positioner ESD Counter Threshold	0x03090051	I32	Ch	R(W)	X	(X)	193
Positioner Target Reached Threshold	0x03090052	I32	Ch	R(W)	X	(X)	194
Positioner Target Hold Threshold	0x03090053	I32	Ch	R(W)	X	(X)	195
Save Positioner Type	0x0309000A	I32	Ch	W	X	-	196
Positioner Write Protection	0x0309000D	I32	Ch	RW	X	-	196
Streaming Properties							
Stream Base Rate	0x040F002C	I32	Dev	RW	-	-	197
Stream External Sync Rate	0x040F002D	I32	Dev	RW	-	-	198
Stream Options	0x040F005D	I32	Dev	RW	-	-	199
Stream Load Maximum	0x040F0301	I32	Dev	R	-	-	200
Diagnostic Properties							
Channel Error	0x0502007A	I32	Ch	R	X	-	200
Channel Temperature	0x05020034	I32	Ch	R	X	-	201
Bus Module Temperature	0x05030034	I32	Mod	R	X	-	202

Continued on next page

Table 4.1 – *Continued from previous page*

Property	Code	Type	Idx	Access	CG[1]	NV[2]	Page
Auxiliary Properties							
Aux Positioner Type	0x0802003C	I32	Ch	RW	X	X	203
Aux Positioner Type Name	0x0802003D	String	Ch	R	-	-	204
Aux Input Select	0x08020018	I32	Ch	RW	X	X	204
Aux I/O Module Input Index	0x081100AA	I32	Ch	RW	X	X	205
Aux Direction Inversion	0x0809000E	I32	Ch	RW	X	X	207
Aux I/O Module Input0 / Input1 Value	0x08110000	I64	Ch	R	X	-	208
Aux I/O Module Input0 / Input1 Value	0x08110001	I64	Ch	R	X	-	208
Aux Digital Input Value	0x080300AD	I32	Mod	R	X	-	208
Aux Digital Output Value / Set / Clear	0x080300AE	I32	Mod	RW	X	-	209
Aux Digital Output Value / Set / Clear	0x080300B0	I32	Mod	W	X	-	209
Aux Digital Output Value / Set / Clear	0x080300B1	I32	Mod	W	X	-	209
Aux Analog Output Value0 / Value1	0x08030000	I32	Mod	RW	X	-	210
Aux Analog Output Value0 / Value1	0x08030001	I32	Mod	RW	X	-	210
I/O Module Properties							
I/O Module Options	0x0603005D	I32	Mod	RW	X	-	211
I/O Module Voltage	0x06030031	I32	Mod	RW	X	-	213
I/O Module Analog Input Range	0x060300A0	I32	Mod	RW	X	X	213
Input Trigger Properties							
Device Input Trigger Mode	0x060D0087	I32	Dev	RW	-	-	215
Device Input Trigger Condition	0x060D005A	I32	Dev	RW	-	-	216
Output Trigger Properties							
Channel Output Trigger Mode	0x060E0087	I32	Ch	RW	X	-	217
Channel Output Trigger Polarity	0x060E005B	I32	Ch	RW	X	-	218
Channel Output Trigger Pulse Width	0x060E005C	I32	Ch	RW	X	-	219

Continued on next page

Table 4.1 – *Continued from previous page*

Property	Code	Type	Idx	Access	CG[1]	NV[2]	Page
Channel Position Compare Start Threshold	0x060E0058	I64	Ch	RW	X	-	220
Channel Position Compare Increment	0x060E0059	I64	Ch	RW	X	-	221
Channel Position Compare Direction	0x060E0026	I32	Ch	RW	X	-	221
Channel Position Compare Limit Min	0x060E0020	I64	Ch	RW	X	-	222
Channel Position Compare Limit Max	0x060E0021	I64	Ch	RW	X	-	223
Hand Control Module Properties							
Hand Control Module Lock Options	0x020C0083	I32	Dev	RW	-	-	224
Hand Control Module Default Lock Options	0x020C0084	I32	Dev	RW	-	X	226
API Properties							
Event Notification Options	0xF010005D	I32	API	RW	-	-	227
Auto Reconnect	0xF01000A1	I32	API	RW	-	-	228

[1] Command Group: This column defines if a property may be used in command groups. See section 2.14 "Command Groups" for more information.

[2] Non-Volatile: This column defines if a property is stored in non-volatile memory. Non-Volatile properties need not be configured on every power-up.

4.2 Device Properties

4.2.1 Number of Channels

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_NUMBER_OF_CHANNELS	0x020F0017	I32	Dev	R	-

ASCII Command: `[:PROPerty]:DEVice:NOCHannels`

Description

This property holds the total number of channels the connected device has. It defines the valid range for channel index parameters. The channel index is zero based. Therefore, the maximum index is *number of channels* - 1.

Note that the number of channels does not represent the number of positioners that are currently connected to the device.

Example

```
SA_CTL_Result_t result;
int32_t channels;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_NUMBER_OF_CHANNELS, &channels, 0
);
if (result == SA_CTL_ERROR_NONE) {
    // 'channels' holds the number of available channels of the device
}
```

See Also

4.2.2 Number of Bus Modules, 4.3.3 Number of Bus Module Channels

4.2.2 Number of Bus Modules

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_NUMBER_OF_BUS_MODULES	0x020F0016	I32	Dev	R	-

ASCII Command: `[:PROPerty]:DEVice:NOBModules`

Description

This property holds the number of modules the connected device has. It defines the valid range for module index parameters. The module index is zero based. Therefore, the maximum index is *number of modules* - 1.

Example

```
SA_CTL_Result_t result;
int32_t modules;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_NUMBER_OF_BUS_MODULES, &modules
);
if (result == SA_CTL_ERROR_NONE) {
    // 'modules' holds the number of available modules of the device
}
```

See Also

4.3.3 Number of Bus Module Channels

4.2.3 Device State

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DEVICE_STATE	0x020F000F	I32	Dev	R	-

ASCII Command: `[:PROPerty]:DEVice:STATe`

Description

This property holds the device state. The value is a bit field containing independent flags. Their meanings are described in section 2.8.1 "Device State Flags".

Undefined flags are reserved for future use. Therefore, the user software should not rely on a static value of undefined flags.

Example

```
SA_CTL_Result_t result;
int32_t state;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_DEVICE_STATE, &state
);
if (result == SA_CTL_ERROR_NONE) {
    // use bit masking to extract the needed information from the state
    if (state & SA_CTL_DEV_STATE_BIT_HM_PRESENT) {
        // a hand controller is connected to the device
    }
}
```

See Also

4.3.2 Module State, 4.4.9 Channel State

4.2.4 Device Serial Number

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DEVICE_SERIAL_NUMBER	0x020F005E	String	Dev	R	-

ASCII Command: `[:PROPerty]:DEVice:SNUMber`

Description

This property may be used to identify a device connected to the PC. Each device has a unique serial number which makes it possible to distinguish one from another. The device serial number consists of the global device name ('MCS2') and an individual number.

Example

```
SA_CTL_Result_t result;
char deviceSn[SA_CTL_STRING_MAX_LENGTH];
size_t ioStrSize = sizeof(deviceSn);
result = SA_CTL_GetProperty_s(
    dHandle,0,SA_CTL_PKEY_DEVICE_SERIAL_NUMBER,deviceSn,&ioStrSize
);
if (result == SA_CTL_ERROR_NONE) {
    // 'deviceSn' holds the serial number string, e.g. 'MCS2-00000001'
```

```
}
```

See Also

4.2.5 Device Name

4.2.5 Device Name

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DEVICE_NAME	0x020F003D	String	Dev	RW	-

ASCII Command: `[:PROPerty]:DEVice:NAME`

Description

This property may be used to identify a device connected to the PC. In contrast to the device serial number, the device name is writable by the user. The name is stored to non-volatile memory. By default, the device name is set to the device serial number string. Note that the device name is *not* reset to its default when performing a firmware update.

Example

```
SA_CTL_Result_t result;
char deviceName[SA_CTL_STRING_MAX_LENGTH];
size_t ioStringSize = sizeof(deviceName);
result = SA_CTL_GetProperty_s(
    dHandle,0,SA_CTL_PKEY_DEVICE_NAME,deviceName,&ioStringSize
);
if (result == SA_CTL_ERROR_NONE) {
    // 'deviceName' holds the user defined name of the device
}
```

See Also

4.2.4 Device Serial Number

4.2.6 Emergency Stop Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_EMERGENCY_STOP_MODE	0x020F0088	I32	Dev	RW	-

ASCII Command: [:PROPerty]:DEVice:ESTop:MODE

Description

This property specifies the emergency stop mode of the device. See section 2.17.2 "Emergency Stop Mode" for more information.

The default value is SA_CTL_EMERGENCY_STOP_MODE_NORMAL (0).

Valid Range

SA_CTL_EMERGENCY_STOP_MODE_NORMAL (0),
SA_CTL_EMERGENCY_STOP_MODE_RESTRICTED (1),
SA_CTL_EMERGENCY_STOP_MODE_AUTO_RELEASE (2)

Example

```
// set emergency stop mode to normal mode
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_EMERGENCY_STOP_MODE,
  SA_CTL_EMERGENCY_STOP_MODE_NORMAL
);
```

See Also

4.13.1 Device Input Trigger Mode

4.2.7 Network Discover Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_NETWORK_DISCOVER_MODE	0x020F0159	I32	Dev	RW	-

ASCII Command: [:PROPerty]:DEVice:NETWork:DISCover:MODE

Description

This property specifies the discover mode for devices with ethernet interface. The discover feature allows to use the SA_CTL_FindDevices function to list devices with ethernet interface without knowing the actual IP address. The MCS2 devices use broadcast packets to inform about their presence in the network and for the discovery mechanism. This technique is quite common for network devices like switches, routers, etc. However, some users might wish to limit the traffic in a restricted network. Therefore, the behavior of the discovery mechanism is configurable.

The following modes are available:

Mode	Name	Short Description
0	SA_CTL_NETWORK_DISCOVER_MODE_DISABLED	The discover feature is disabled. No broadcast packets will be generated. Devices will not be found by the SA_CTL_FindDevices function.
1	SA_CTL_NETWORK_DISCOVER_MODE_PASSIVE	The device will not generate packets to inform about its presence but still reacts to direct discover requests.
2	SA_CTL_NETWORK_DISCOVER_MODE_ACTIVE	The device informs about its presence and reacts to all discover requests.

See section 2.1 "Connecting and Disconnecting" for more information.

The default value is SA_CTL_NETWORK_DISCOVER_MODE_ACTIVE (2). This property is stored to non-volatile memory and need not be configured on every power-up.

Example

```
// disable the network discover feature
result = SA_CTL_SetProperty_i32(
  dHandle,
  0,
  SA_CTL_PKEY_NETWORK_DISCOVER_MODE,
  SA_CTL_NETWORK_DISCOVER_MODE_DISABLED
```

```
);
```

See Also

3.2.4 SA_CTL_FindDevices

4.3 Module Properties

4.3.1 Power Supply Enabled

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POWER_SUPPLY_ENABLED	0x02030010	I32	Mod	RW	X

ASCII Command: [:PROPerty]:MODule#:PSUPply[:ENABled]

Description

This property enables or disables the positioner driver power supply of the module. Of course the power supply must be enabled to perform positioner movements. Otherwise, if a movement is commanded, the SA_CTL_EVENT_MOVEMENT_FINISHED event that is generated by the channel will hold a SA_CTL_ERROR_POWER_SUPPLY_DISABLED error as parameter.

The default value is SA_CTL_ENABLED (0x01).

Valid Range

SA_CTL_DISABLED (0x00), SA_CTL_ENABLED (0x01)

Example

```
// switch off the driver power supply of the first module
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POWER_SUPPLY_ENABLED, SA_CTL_DISABLED
);
```

See Also

4.4.1 Amplifier Enabled

4.3.2 Module State

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_MODULE_STATE	0x0203000F	I32	Mod	R	X

ASCII Command: `[:PROPerty]:MODule#:STATe`

Description

This property holds the module state. The value is a bit field containing independent flags. Their meanings are described in section 2.8.2 "Module State Flags".

Undefined flags are reserved for future use. Therefore, the user software should not rely on a static value of undefined flags.

Example

```
SA_CTL_Result_t result;
int32_t state;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_MODULE_STATE, &state
);
if (result == SA_CTL_ERROR_NONE) {
    // use bit masking to extract the needed information from the state
    if (state & SA_CTL_MOD_STATE_BIT_SM_PRESENT) {
        // a sensor module is connected to the module
    }
}
```

See Also

4.2.3 Device State, 4.4.9 Channel State

4.3.3 Number of Bus Module Channels

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_NUMBER_OF_BUS_MODULE_CHANNELS	0x02030017	I32	Mod	R	X

ASCII Command: `[:PROPerty]:MODule#:NOMChannels`

Description

This property holds the number of channels the addressed module has.

Example

```
SA_CTL_Result_t result;
int32_t modChannels;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_NUMBER_OF_BUS_MODULE_CHANNELS, &modChannels
);
if (result == SA_CTL_ERROR_NONE) {
    // 'modChannels' holds the number of channel of the module 0
}
```

See Also

4.2.2 Number of Bus Modules, 4.2.1 Number of Channels

4.4 Positioner Properties

4.4.1 Amplifier Enabled

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AMPLIFIER_ENABLED	0x0302000D	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:AMPLifier[:ENABled]`

Description

This property enables or disables the positioner driver amplifier of the channel. Of course, the amplifier must be enabled to perform positioner movements. Otherwise, if a movement is commanded, the SA_CTL_EVENT_MOVEMENT_FINISHED event that is generated by the channel will hold a SA_CTL_ERROR_AMPLIFIER_DISABLED error as parameter.

The default value is SA_CTL_ENABLED (0x01).

Valid Range

SA_CTL_DISABLED (0x00), SA_CTL_ENABLED (0x01)

Example

```
// switch off the driver power amplifier of the first channel
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_AMPLIFIER_ENABLED, SA_CTL_DISABLED
);
```

See Also

4.3.1 Power Supply Enabled

4.4.2 Positioner Control Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POSITIONER_CONTROL_OPTIONS	0x0302005D	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:PCONtrol:OPTions

Description

This property defines several positioner control related options. The value is a bit field containing independent flags. The following flags are available:

Bit	C-Definition	Code
0	SA_CTL_POS_CTRL_OPT_BIT_ACC_REL_POS_DIS	0x00000001
1	SA_CTL_POS_CTRL_OPT_BIT_NO_SLIP	0x00000002
2	SA_CTL_POS_CTRL_OPT_BIT_NO_SLIP_WHILE_HOLDING	0x00000004
3	SA_CTL_POS_CTRL_OPT_BIT_FORCED_SLIP_DIS	0x00000008
4	SA_CTL_POS_CTRL_OPT_BIT_STOP_ON_FOLLOWING_ERR	0x00000010
5	SA_CTL_POS_CTRL_OPT_BIT_TARGET_TO_ZERO_VOLTAGE	0x00000020

Undefined flags are reserved and should be set to zero. See section 2.6.4 "Closed-Loop Movements" for a more detailed description of the positioner control options flags.

This property is stored to nonvolatile memory and need not be configured on every power-up. The default value is 0 (all flags cleared).

Example

```
// enable the "no-slip-while-holding" feature for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle,
    0,
    SA_CTL_PKEY_POSITIONER_CONTROL_OPTIONS,
    SA_CTL_POS_CTRL_OPT_BIT_NO_SLIP_WHILE_HOLDING
);
```

See Also

4.4.8 Move Mode, 4.4.3 Actuator Mode

4.4.3 Actuator Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_ACTUATOR_MODE	0x03020019	I32	Ch	RW	-

ASCII Command: `[:PROPerty]:CHANnel#:ACTuator:MODE`

Description

This property specifies the type of driving signal generation. See section 2.6.4 "Closed-Loop Movements" for a more detailed description of the actuator modes. It is not allowed to change the actuator mode during an ongoing movement. In that case a `SA_CTL_ERROR_BUSY_MOVING` error is returned.

Note that the *low vibration* mode requires the velocity and acceleration control to be active. If the velocity control is not already enabled (move velocity != 0), the move velocity is set implicitly to a default velocity of 10×10^9. If the acceleration control is not already enabled (move acceleration != 0), the move acceleration is set implicitly to a default acceleration of 100×10^9.

Note that all referencing movements are performed with the normal mode even if this property is configured to `SA_CTL_ACTUATOR_MODE_LOW_VIBRATION`.

The default mode is `SA_CTL_ACTUATOR_MODE_NORMAL` (0).

Valid Range

`SA_CTL_ACTUATOR_MODE_NORMAL` (0),
`SA_CTL_ACTUATOR_MODE_QUIET` (1),
`SA_CTL_ACTUATOR_MODE_LOW_VIBRATION` (2)

NOTICE
The low vibration actuator mode needs a feature permission to be activated on the controller. See section 2.19 "Feature Permissions" for more information.

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 0;
// configure the 'quiet' actuator mode for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle,
    channelIdx,
    SA_CTL_PKEY_ACTUATOR_MODE,
    SA_CTL_ACTUATOR_MODE_QUIET
);
```

See Also

4.4.15 Move Velocity, 4.4.16 Move Acceleration, 4.4.8 Move Mode, 4.4.2 Positioner Control Options

4.4.4 Control Loop Input

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CONTROL_LOOP_INPUT	0x03020018	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:CLINput

Description

This property specifies which signal is used as input for the control-loop. For the majority of applications this property will be set to SA_CTL_CONTROL_LOOP_INPUT_SENSOR, meaning the integrated sensor of the positioner is used as feedback signal for the control-loop.

Nonetheless it is also possible to use external signals. E.g. an analog voltage derived from a force sensor can be feed into an analog input of the MCS2 I/O module to implement a force feedback control for a gripper. Set this property to SA_CTL_CONTROL_LOOP_INPUT_AUX_IN to use one of the auxiliary inputs as control-loop feedback. Please refer to section 2.16.5 "Using Analog Inputs as Control-Loop Feedback" for more information on the auxiliary configuration.

In some cases it may be useful to prohibit the closed-loop operation of a channel. This can be achieved by setting this property to SA_CTL_CONTROL_LOOP_INPUT_DISABLED.
A SA_CTL_ERROR_CONTROL_LOOP_INPUT_DISABLED error will be generated when trying to command a closed-loop movement in this case.

The default input is SA_CTL_CONTROL_LOOP_INPUT_SENSOR (1). This property is stored to non-volatile memory and need not be configured on every power-up.

Valid Range

SA_CTL_CONTROL_LOOP_INPUT_DISABLED (0),
SA_CTL_CONTROL_LOOP_INPUT_SENSOR (1),
SA_CTL_CONTROL_LOOP_INPUT_AUX_IN (2)

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 0;
// configure the sensor as input for the control-loop for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle,
    channelIdx,
    SA_CTL_PKEY_CONTROL_LOOP_INPUT,
    SA_CTL_CONTROL_LOOP_INPUT_SENSOR
);
```

See Also

4.4.5 Sensor Input Select

4.4.5 Sensor Input Select

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SENSOR_INPUT_SELECT	0x0302009D	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:CLINput

Description

This property specifies which sensor signal is used for the 'sensor' input of the control-loop input mux. (See Control Loop Input property.) The property is only relevant if a SmarAct PicoScale laser interferometer is connected as sensor module. The PicoScale calculation system can perform various calculations with different values of the device, in particular even from different channels. The calculation system may then be used to generate a control-loop input signal for the MCS2 channel. Set this property to SA_CTL_SENSOR_INPUT_SELECT_CALC_SYS to configure the calculation system. Please refer to section 2.10 "PicoScale Sensor Module" and figure 2.10 "Auxiliary Input Configuration (per channel)" for more information.

The default input is SA_CTL_SENSOR_INPUT_SELECT_POSITION (0). This property is stored to non-volatile memory and need not be configured on every power-up.

Valid Range

SA_CTL_SENSOR_INPUT_SELECT_POSITION (0),
SA_CTL_SENSOR_INPUT_SELECT_CALC_SYS (1)

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 0;
// configure the PSC calculation system as input
// for the control-loop for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle,
    channelIdx,
    SA_CTL_PKEY_SENSOR_INPUT_SELECT,
    SA_CTL_SENSOR_INPUT_SELECT_CALC_SYS
);
```

See Also

4.4.4 Control Loop Input

4.4.6 Positioner Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POSITIONER_TYPE	0x0302003C	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:PTYPe[:CODE]

Description

This property is used to tell the channel what type of positioner is connected. The type implicitly gives the controller information about how to calculate positions, handle the referencing and configure the control-loop.

Each channel stores this setting to non-volatile memory. Consequently, there is no need to set this property on every initialization. If the positioner type of a channel is changed, the positioner is stopped implicitly. Furthermore the calibration becomes invalid and the physical position becomes unknown. (The Channel State bits SA_CTL_CH_STATE_BIT_IS_CALIBRATED and SA_CTL_CH_STATE_BIT_IS_REFERENCED are reset to zero.)

Note that SA_CTL_Calibrate must be called to ensure proper operation of the positioner if the positioner type was changed.

See section 2.5 "Positioner Types" for more information on positioner types.

Valid Range

Please refer to the *MCS2 Positioner Types* document for a list of valid positioner type codes.

Example

```
// set the positioner type 'SLxS1SS' (type code 300) for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POSITIONER_TYPE, 300
);
```

4.4.7 Positioner Type Name

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POSITIONER_TYPE_NAME	0x0302003D	String	Ch	R	-

ASCII Command: [:PROPerty]:CHANnel#:PTYPe:NAME

Description

This property holds a descriptive name of the configured positioner type. The positioner type name is a null terminated string. Note that the name is read-only.

Example

```
SA_CTL_Result_t result;
char name[SA_CTL_STRING_MAX_LENGTH];
size_t ioStringSize = sizeof(name);
result = SA_CTL_GetProperty_s(
    dHandle, 0, SA_CTL_PKEY_POSITIONER_TYPE_NAME, name, &ioStringSize
);
if (result == SA_CTL_ERROR_NONE) {
    // 'name' holds the name of the configured positioner type
}
```

See Also

4.4.6 Positioner Type

4.4.8 Move Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_MOVE_MODE	0x03050087	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:MMODe

Description

This property specifies which movement mode is used when commanding a positioner movement using SA_CTL_Move. Depending on the configured move mode the *move value* parameter of the SA_CTL_Move function is interpreted differently. See section 2.6.3 "Open-Loop Movements" and 2.6.4 "Closed-Loop Movements" for a description of all related properties for the different move modes.

The default mode is SA_CTL_MOVE_MODE_CL_ABSOLUTE (0).

Valid Range

SA_CTL_MOVE_MODE_CL_ABSOLUTE (0),
SA_CTL_MOVE_MODE_CL_RELATIVE (1),
SA_CTL_MOVE_MODE_SCAN_ABSOLUTE(2),
SA_CTL_MOVE_MODE_SCAN_RELATIVE(3),
SA_CTL_MOVE_MODE_STEP (4)

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 0;
// configure an open-loop step movement with full amplitude at 2kHz
result = SA_CTL_SetProperty_i32(
    dHandle,channelIdx, SA_CTL_PKEY_MOVE_MODE, SA_CTL_MOVE_MODE_STEP
);
if (result) {// handle error, abort}
result = SA_CTL_SetProperty_i32(
    dHandle,channelIdx, SA_CTL_PKEY_STEP_AMPLITUDE, 65535
);
if (result) {// handle error, abort}
result = SA_CTL_SetProperty_i32(
    dHandle,channelIdx, SA_CTL_PKEY_STEP_FREQUENCY, 2000
);
if (result == SA_CTL_ERROR_NONE) {
    // perform 100 steps
    result = SA_CTL_Move(
        dHandle, channelIdx, 100
    );
}
```

4.4.9 Channel State

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CHANNEL_STATE	0x0305000F	I32	Ch	R	X

ASCII Command: `[:PROPerty]:CHANnel#:STATe`

Description

This property holds the channel state. The value is a bit field containing independent flags. Their meaning is described in section 2.8.3 "Channel State Flags".

Undefined flags are reserved for future use. Therefore, the user software should not rely on a static value of undefined flags.

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 0;
int32_t state;
result = SA_CTL_GetProperty_i32(
    dHandle, channelIdx, SA_CTL_PKEY_CHANNEL_STATE, &state
);
if (result == SA_CTL_ERROR_NONE) {
    // use bit masking to determine the channels movement state
    if ((state & (SA_CTL_CH_STATE_BIT_ACTIVELY_MOVING |
                  SA_CTL_CH_STATE_BIT_CLOSED_LOOP_ACTIVE)) == 0) {
        // positioner is stopped
    }
}
```

See Also

4.2.3 Device State, 4.3.2 Module State

4.4.10 Position

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POSITION	0x0305001D	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:POSition[:CURRent]

Description

This property holds the current position of a positioner. Note that it can only be used for positioners that have a sensor attached to it. To determine if a sensor is present the Channel State bit SA_CTL_CH_STATE_BIT_SENSOR_PRESENT may be polled.

The interpretation of the read position value depends on the configured positioner type. The unit is pico meter (pm) for linear positioners and nano degree (n°) for rotatory positioners.

Read the Positioner Base Unit property to distinguish between linear and rotatory positioner type.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$ pm or n°.

Example

```
SA_CTL_Result_t result;
int64_t position;
result = SA_CTL_GetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_POSITION, &position
);
if (result == SA_CTL_ERROR_NONE) {
    // `position` holds the current position of channel 0
}
```

See Also

4.8.3 Positioner Base Unit, 4.8.4 Positioner Base Resolution

4.4.11 Target Position

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_TARGET_POSITION	0x0305001E	I64	Ch	R	X

ASCII Command: [:PROPerty]:CHANnel#:POSition:TARGet

Description

This property holds the target position of a positioner.

See Also

4.4.10 Position

4.4.12 Scan Position

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SCAN_POSITION	0x0305001F	I64	Ch	R	X

ASCII Command: [:PROPerty]:CHANnel#:POSition:SCAN

Description

This property holds the current scan position of a positioner. The scan position represents the voltage level that is currently applied to the piezo element of a positioner.

This property is mainly of interest when using the SA_CTL_MOVE_MODE_SCAN_ABSOLUTE and SA_CTL_MOVE_MODE_SCAN_RELATIVE Move Modes, since these modes are used to control the scan position.

The scan position is given in 16-bit increments from 0 . . . 65 535, where 0 corresponds to 0 V and 65 535 to 100 V.

Example

```
SA_CTL_Result_t result;
int64_t position;
result = SA_CTL_GetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_SCAN_POSITION, &position
);
if (result == SA_CTL_ERROR_NONE) {
    // 'position' holds the current scan position of channel 0
}
```

See Also

4.4.13 Scan Velocity, 4.4.8 Move Mode

4.4.13 Scan Velocity

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SCAN_VELOCITY	0x0305002A	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:SCAN:VELocity

Description

This property specifies the scan velocity of a positioner. The scan velocity is given in 16-bit increments per second. With a value of 1 a scan over the full range from 0 to 65 535 takes 65 535 seconds while at maximum velocity the scan is performed in one micro second.

To perform a scan movement via the SA_CTL_Move function, the Move Mode property must be set to SA_CTL_MOVE_MODE_SCAN_ABSOLUTE or SA_CTL_MOVE_MODE_SCAN_RELATIVE first.

The default value is 65 535.

Valid Range

1 . . . 65 535 000 000

Example

```
// set the scan velocity for channel 0
// (full range scan in 1 second)
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_SCAN_VELOCITY, 65535
);
```

See Also

4.4.12 Scan Position, 4.4.8 Move Mode

4.4.14 Hold Time

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_HOLD_TIME	0x03050028	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:HOLDtime

Description

This property specifies how long (in ms) the position is actively held after reaching the target position. After the hold time elapsed the channel is stopped.

The Channel State bit SA_CTL_CH_STATE_BIT_CLOSED_LOOP_ACTIVE will be read as one as long as the the position is actively held.

A value of 0 deactivates this feature, a value of SA_CTL_INFINITE (0xffffffff) sets the channel to infinite holding. (until manually stopped with SA_CTL_Stop).

Note that the end stop detection is still active in holding state. If a positioner is moved away from the target position by external forces and the channel is not able to hold the target position for a longer time an end stop is triggered. A SA_CTL_EVENT_HOLDING_ABORTED event is generated to notify about this and the channel is stopped.

The default hold time is SA_CTL_INFINITE.

Valid Range

0 ... 0xffffffff

Example

```
// set hold time for channel 0 to infinite holding
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_HOLD_TIME, SA_CTL_INFINITE
);
```

See Also

4.4.8 Move Mode

4.4.15 Move Velocity

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_MOVE_VELOCITY	0x03050029	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:VELocity

Description

This property specifies the velocity of a positioner for closed-loop movement commands. The value is given in pm s^{-1} for linear positioners and in n° s^{-1} for rotary positioners. If a velocity > 0 is configured, all following closed-loop movement commands will be executed with velocity control.

Note that the channel will not drive the positioner with frequencies above the maximum allowed frequency (see Max Closed Loop Frequency). If the maximum frequency is set too low for a certain velocity, then the velocity might not be reached or held since the driver will cap at the maximum driving frequency. In this case increase the maximum frequency.

Note that the move velocity also applies to movements executed during the find reference sequence (see SA_CTL_Reference).

The default value is 0, meaning that the velocity control is inactive. In this state the behavior of closed-loop commands is influenced by the maximum driving frequency (see Max Closed Loop Frequency).

It is not allowed to *enable* or *disable* the velocity control during an ongoing movement. In that case a SA_CTL_ERROR_BUSY_MOVING error is returned. Anyway, *modifying* the velocity of an ongoing movement is possible.

Valid Range

$0 \ldots 100 \times 10^9$

Example

```
// enable velocity control by configuring 1mm/s for channel 0
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_MOVE_VELOCITY, 1e9
);
```

See Also

4.4.16 Move Acceleration, 4.4.8 Move Mode, 4.4.17 Max Closed Loop Frequency

4.4.16 Move Acceleration

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_MOVE_ACCELERATION	0x0305002B	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:ACCeleration

Description

This property specifies the acceleration of a positioner for closed-loop movement commands. The value is given in pm s^{-2} for linear positioner and in n° s^{-2} for rotary positioners. If an acceleration > 0 is configured, all following closed-loop movement commands will be executed with acceleration control. The acceleration control requires the velocity control to be enabled (Move Velocity > 0).

Note that the move acceleration also applies to movements executed during the find reference sequence (see SA_CTL_Reference).

The default value is 0, meaning that the acceleration control is inactive.

It is not allowed to *enable* or *disable* the acceleration control during an ongoing movement. In that case a SA_CTL_ERROR_BUSY_MOVING error is returned. Anyway, *modifying* the acceleration of an ongoing movement is possible.

NOTICE

For closed-loop movements with enabled acceleration control a SA_CTL_Stop command instructs the positioner to come to a halt by decelerating to zero velocity. A second "stop" command triggers a hard stop.

Valid Range

$0 \ldots 10 \times 10^{12}$

Example

```
// enable acceleration control by configuring 1mm/s2 for channel 0
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_MOVE_ACCELERATION, 1e9
);
```

See Also

4.4.15 Move Velocity, 4.4.8 Move Mode

4.4.17 Max Closed Loop Frequency

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_MAX_CL_FREQUENCY	0x0305002F	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:MCLFrequency[:CURRent]`

Description

This property specifies the maximum frequency that the positioner is driven with when issuing closed-loop movement commands.

The maximum allowed frequency depends on the actual positioner as well as the environment. (E.g. HV and UHV environment requires lower allowed frequencies.)

This property is not held in non-volatile memory but the default value at device startup is configurable (see Default Max Closed Loop Frequency).

Valid Range

50 ... 20 000 Hz

Example

```
// set maximum closed-loop frequency to 3kHz for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_MAX_CL_FREQUENCY, 3000
);
```

See Also

4.4.18 Default Max Closed Loop Frequency

4.4.18 Default Max Closed Loop Frequency

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DEFAULT_MAX_CL_FREQUENCY	0x03050057	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:MCLFrequency:DEFault`

Description

This property specifies the default value at device startup for the maximum closed-loop frequency.

Valid Range

50 . . . 20 000 Hz

Example

```
// set default maximum closed-loop frequency
// at start up to 6kHz for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_DEFAULT_MAX_CL_FREQUENCY, 6000
);
```

See Also

4.4.17 Max Closed Loop Frequency

4.4.19 Step Frequency

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_STEP_FREQUENCY	0x0305002E	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:STEP:FREQuency

Description

This property specifies the frequency in Hz that open-loop steps are performed with. To perform open-loop steps by using the SA_CTL_Move function, the Move Mode property must be set to SA_CTL_MOVE_MODE_STEP first. See section 2.6.3 "Open-Loop Movements" for more information.

The default frequency is 1000 Hz.

Valid Range

1 . . . 20 000 Hz

Example

```
// set the step frequency to 1kHz for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_STEP_FREQUENCY, 1000
);
```

See Also

4.4.20 Step Amplitude, 4.4.8 Move Mode

4.4.20 Step Amplitude

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_STEP_AMPLITUDE	0x03050030	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:STEP:AMPLitude

Description

This property specifies the amplitude that open-loop steps are performed with. The Move Mode property must be set to `SA_CTL_MOVE_MODE_STEP` first, before open-loop steps may be performed with the `SA_CTL_Move` function. See section 2.6.3 "Open-Loop Movements" for more information.

Lower amplitude values result in a smaller step width. The step amplitude is a 16bit value from 1 . . . 65 535, where 65 535 corresponds to 100 V.

The default amplitude is 65 535 (100 V).

Valid Range

1 . . . 65 535

Example

```
// set the step amplitude to maximum (100V) for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_STEP_AMPLITUDE, 65535
);
```

See Also

4.4.19 Step Frequency, 4.4.8 Move Mode

4.4.21 Following Error Limit

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_FOLLOWING_ERROR_LIMIT	0x03050055	I64	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:FELimit`

Description

This property specifies the limit for the following error detection. The value is given in pm for linear positioners and in n° for rotary positioners. Setting the following error limit to zero disables the detection. Please refer to section 2.11 "Following Error Detection" for more information.

This property is stored to non-volatile memory and need not be configured on every power-up. The default value is 0 (disabled).

Valid Range

$0 \ldots 100 \times 10^{12}$

Example

```
// set following error limit to 100um for channel 0
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_FOLLOWING_ERROR_LIMIT, 100000000
);
```

See Also

4.4.2 Positioner Control Options, 4.4.15 Move Velocity

4.4.22 Broadcast Stop Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_BROADCAST_STOP_OPTIONS	0x0305005D	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:BSTop:OPTions`

Description

This property specifies the behavior of a broadcast stop of a channel. It is typically useful when multiple channels are moving simultaneously and an end stop (or range limit) on one channel should cause a halt on all other channels. Please refer to section 2.13 "Stop Broadcasting" for more information.

The value is a bit field containing independent flags with the following meaning:

Bit	Name	Short Description
0	SA_CTL_STOP_OPT_BIT_END_STOP_REACHED	Broadcast stop command if a mechanical end stop was detected.
1	SA_CTL_STOP_OPT_BIT_RANGE_LIMIT_REACHED	Broadcast stop command if a range limit was reached.
2	SA_CTL_STOP_OPT_BIT_FOLLOWING_LIMIT_REACHED	Broadcast stop command if a following error limit was exceeded.

Undefined flags are unused but might get a meaning in future updates. Undefined flags should be set to zero.

The default value is 0.

Example

```
// enable stop broadcasting of channel 0 for end stops and range limits
result = SA_CTL_SetProperty_i32(
    dHandle,
    0,
    SA_CTL_PKEY_BROADCAST_STOP_OPTIONS,
      (SA_CTL_STOP_OPT_BIT_END_STOP_REACHED |
       SA_CTL_STOP_OPT_BIT_RANGE_LIMIT_REACHED)
);
```

4.4.23 Sensor Power Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SENSOR_POWER_MODE	0x03080019	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:SENSor:PMODe

Description

This property specifies the sensor power mode. It may be used to activate or deactivate the sensor that is attached to the positioner. It effectively turns the power supply of the sensor on or off.

Please refer to section 2.9 "Sensor Power Modes" for more information on the sensor power modes.

This property is stored to non-volatile memory and need not be configured on every power-up.

The following sensor power modes are available:

Mode	Name	Short Description
0	SA_CTL_SENSOR_MODE_DISABLED	The sensor power supply is turned off continuously.
1	SA_CTL_SENSOR_MODE_ENABLED	The sensor is continuously supplied with power.
2	SA_CTL_SENSOR_MODE_POWER_SAVE	The sensor power supply is pulsed to keep the heat generation low.

Example

```
// set power save mode for the sensor of channel 0
result = SA_CTL_SetProperty_i32(
    dHandle,0,SA_CTL_PKEY_SENSOR_POWER_MODE,SA_CTL_SENSOR_MODE_POWER_SAVE
);
```

See Also

4.4.24 Sensor Power Save Delay

4.4.24 Sensor Power Save Delay

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SENSOR_POWER_SAVE_DELAY	0x03080054	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:SENSor:PSDelay

Description

This property specifies the time in ms before the channel disables the sensor after a movement has finished. It has no meaning if the Sensor Power Mode is not configured to power save mode. In power save mode the sensor is disabled most of the time. Before a movement can be started it must be enabled by the channel to keep track of the current position. Once the movement has finished the sensor can be disabled again. The sensor power save delay configures an additional delay before the sensor power is disabled. If a new movement is started while this delay is running, the sensor is still enabled and the movement can be started directly. Since it takes a few milliseconds to enable the sensor, this setting may be used to optimize the timing of a movement sequence.

Please refer to section 2.9 "Sensor Power Modes" for more information on the sensor power save mode.

This property is stored to non-volatile memory and need not be configured on every power-up. The default value is 100 ms.

Valid Range

0 . . . 5000

Example

```
// set power save delay for the sensor of channel 0 to 200 ms
result = SA_CTL_SetProperty_i32(
    dHandle,0,SA_CTL_PKEY_SENSOR_POWER_SAVE_DELAY,200
);
```

See Also

4.4.23 Sensor Power Mode

4.4.25 Position Mean Shift

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POSITION_MEAN_SHIFT	0x03090022	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:POSition:MSHift

Description

This property specifies the filter averaging factor for the position mean filter. The averaging factor must be set as a left-shift value by a power of two. Thus the resulting averaging factor may be calculated by the formula: factor = $2^{meanShift}$.

This property is stored to non-volatile memory and need not be configured on every power-up. The default value is 5 (32-fold position averaging).

Valid Range

0 ... 7

Example

```
// set position mean filter to 0 (disabled) for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POSITION_MEAN_SHIFT, 0
);
```

4.4.26 Safe Direction

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SAFE_DIRECTION	0x03090027	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:SDIRection

Description

This property specifies the safe direction used for calibration and referencing of positioner types that are referenced via a mechanical end stop.

Some positioners are not equipped with a physical reference mark. For these positioners a mechanical end stop is used as a reference point when calling SA_CTL_Reference. Which end stop is used is configured by the safe direction as well as the current Logical Scale Inversion. This should be the direction in which the positioner may safely move without endangering the physical setup of your manipulator system. Since the end stop must be calibrated before it can be properly used as a reference point, the direction settings also affect the behavior of SA_CTL_Calibrate. Positioners that are referenced via an end stop also move to the configured end stop as part of the calibration routine. This movement will use the configured Move Velocity and Move Acceleration.

Please note that the SA_CTL_Reference and SA_CTL_Calibrate functions will ignore their configured start directions for positioners that are referenced via a mechanical end stop and will implicitly use the direction configured by the safe direction and Logical Scale Inversion instead. Please refer to the *MCS2 Positioner Types* document for a list of available positioner types and their reference marks.

Note that when changing the safe direction the positioner must be calibrated again for proper operation.

This property is stored to non-volatile memory and need not be configured on every power-up.

Valid Range

SA_CTL_FORWARD_DIRECTION (0x00), SA_CTL_BACKWARD_DIRECTION (0x01)

Example

```
// set safe direction to forward for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_SAFE_DIRECTION, SA_CTL_FORWARD_DIRECTION
);
```

4.4.27 Control Loop Input Sensor Value

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CL_INPUT_SENSOR_VALUE	0x0302001D	I64	Ch	R	X

ASCII Command: [:PROPerty]:CHANnel#:CLINput:SENSor[:VALue]

Description

This property always returns the ‘sensor’ value regardless of the configured control-loop input. Note that an error is returned if no sensor module or no sensor is present. Please refer to section 2.16.5 "Using Analog Inputs as Control-Loop Feedback" for more information.

Example

```
SA_CTL_Result_t result;
int64_t val;
result = SA_CTL_GetProperty_i64(
```

```
    dHandle, 0, SA_CTL_PKEY_CL_INPUT_SENSOR_VALUE, &val
);
if (result == SA_CTL_ERROR_NONE) {
    // 'val' holds the current sensor position of channel 0
}
```

See Also

4.4.28 Control Loop Input Aux Value

4.4.28 Control Loop Input Aux Value

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CL_INPUT_AUX_VALUE	0x030200B2	I64	Ch	R	X

ASCII Command: [:PROPerty]:CHANnel#:CLINput:AUXiliary[:VALue]

Description

This property always returns the 'auxiliary input' value regardless of the configured control-loop input. Note that an error is returned if no sensor module or no I/O module is available (depending on the configured Aux Input Select property). Please refer to section 2.16.5 "Using Analog Inputs as Control-Loop Feedback" for more information on using auxiliary inputs.

Example

```
SA_CTL_Result_t result;
int64_t val;
result = SA_CTL_GetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_CL_INPUT_AUX_VALUE, &val
);
if (result == SA_CTL_ERROR_NONE) {
    // 'val' holds the auxiliary input value of channel 0
}
```

See Also

4.4.27 Control Loop Input Sensor Value

4.4.29 Target To Zero Voltage Hold Threshold

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_TARGET_TO_ZERO_VOLTAGE_HOLD_TH	0x030200B9	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TTZVoltage:THReshold[:HOLD]`

Description

This property specifies the hold threshold in pm or n° for the target-to-zero-voltage feature. The threshold defines the maximum allowed remaining position error (distance to the target position) for the sequence to terminate. As a guiding value the threshold should be in the range of about ten times the target reached threshold of the configured positioner type but could be also much lower in the particular case. If the threshold is too low the sequence will not terminate.

If a Hold Time is specified the sequence is repeated whenever the difference between current position and target position exceeds the configured threshold. After the hold time elapsed the last sequence is still finished and the channel is stopped.

Note that the target-to-zero-voltage feature must be enabled by setting the `SA_CTL_POS_CTRL_OPT_BIT_TARGET_TO_ZERO_VOLTAGE` flag of the Positioner Control Options property. It has no meaning if the target-to-zero-voltage feature is disabled. If this property is set to 0 the hold threshold value is derived from the Positioner Target Reached Threshold parameter of the configured positioner type.

This property is stored to non-volatile memory and need not be configured on every power-up. The default value is 0.

Please refer to section 2.6.4 "Closed-Loop Movements" for more information on the target-to-zero-voltage feature.

Valid Range

$0 \ldots 10 \times 10^6$.

Example

```
// set the target to zero voltage hold threshold to 25nm for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_TARGET_TO_ZERO_VOLTAGE_HOLD_TH, 25000
);
```

See Also

4.4.2 Positioner Control Options

4.5 Scale Properties

4.5.1 Logical Scale Offset

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_LOGICAL_SCALE_OFFSET	0x02040024	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:LSCale:OFFset

Description

This property specifies the logical scale offset. The value is given in pm for linear positioners and in n° for rotary positioners. It is used to define the relation between the physical and the logical scale. The logical scale offset can be set directly with this property but is also updated by setting the Position property. Please refer to section 2.7.5 "Shifting the Measuring Scale" for more information on defining positions.

This property is stored to non-volatile memory. The default value is 0.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$

Example

```
// set the scale shift of channel 0 to +1mm relative
// to the physical scale
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_LOGICAL_SCALE_OFFSET, 1e9
);
```

See Also

4.5.2 Logical Scale Inversion

4.5.2 Logical Scale Inversion

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_LOGICAL_SCALE_INVERSION	0x02040025	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:LSCale:INVersion`

Description

This property specifies the logical scale inversion. It is used to define the count direction of the logical scale relative to the physical scale. Note that the scale inversion should be defined before the absolute position is determined with the `SA_CTL_Reference` function.

Further note that only the logical scale will be inverted. The Safe Direction setting will not be changed. Thus Positioners With Endstop Reference will move in the opposite direction when executing SA_CTL_Calibrate or SA_CTL_Reference.

Please refer to section 2.7.5 "Shifting the Measuring Scale" for more information on defining positions.

This property is stored to non-volatile memory. The default value is `SA_CTL_NON_INVERTED` (`0x00`).

Valid Range

`SA_CTL_NON_INVERTED` (`0x00`), `SA_CTL_INVERTED` (`0x01`)

Example

```
// enable the scale inversion for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_LOGICAL_SCALE_INVERSION, SA_CTL_INVERTED
);
```

See Also

4.5.1 Logical Scale Offset

4.5.3 Range Limit Min

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_RANGE_LIMIT_MIN	0x02040020	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:RLIMit:MIN

Description

This property specifies the software range limit minimum position. Note that the Range Limit Max must be set to a higher value than the Range Limit Min to enable the limit check.

Please refer to section 2.12 "Software Range Limit" for more information on software range limits.

The default value is 0.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$

Example

```
// set the min range limit to -10mm for channel 0
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_RANGE_LIMIT_MIN, -10000000000
);
```

See Also

4.5.4 Range Limit Max

4.5.4 Range Limit Max

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_RANGE_LIMIT_MAX	0x02040021	I64	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:RLIMit:MAX

Description

This property specifies the software range limit maximum position. Note that the Range Limit Max must be set to a higher value than the Range Limit Min to enable the limit check.

Please refer to section 2.12 "Software Range Limit" for more information on software range limits.

The default value is 0.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$

Example

```
// set the max range limit to +10mm for channel 0
result = SA_CTL_SetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_RANGE_LIMIT_MAX, 10000000000
);
```

See Also

4.5.3 Range Limit Min

4.6 Calibration Properties

4.6.1 Calibration Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CALIBRATION_OPTIONS	0x0306005D	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:CALibration:OPTions`

Description

This property specifies the calibration options. It is used to define the behavior of the calibration routine when calling the `SA_CTL_Calibrate` function.

The value is a bit field containing independent flags. Please refer to section 2.6.1 "Calibrating" for more information on the calibration sequence. Undefined flags are reserved for future use. These flags should be set to zero.

The default value is 0 (all flags cleared).

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 1;
// set calibration options of channel 1 (signal correction sequence)
result = SA_CTL_SetProperty_i32(
    dHandle, channelIdx, SA_CTL_PKEY_CALIBRATION_OPTIONS, 0
);
if (result == SA_CTL_ERROR_NONE) {
    // start signal correction calibration sequence
    result = SA_CTL_Calibrate(dHandle, channelIdx, 0);
}
```

4.6.2 Signal Correction Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_SIGNAL_CORRECTION_OPTIONS	0x0306001C	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:SCORrection:OPTions`

Description

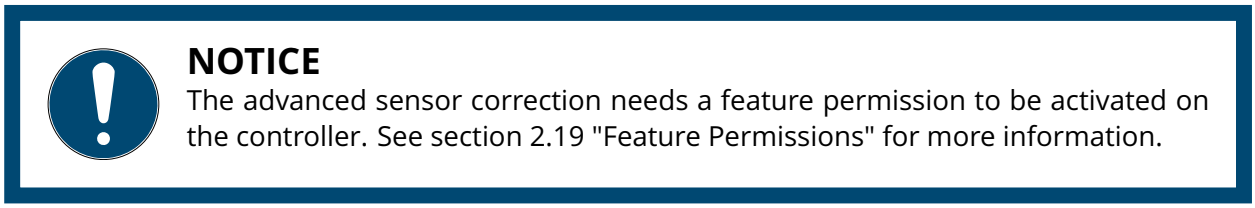
This property specifies the sensor signal correction options. The value is a bit field containing independent flags with the following meaning:

Bit	Name	Short Description
0	Reserved	This bit is reserved and always read as one.
1	Dynamic Amplitude Error Correction	Enables the dynamic sensor amplitude error correction.
2	Reserved	This bit is reserved and always read as one.
3	Dynamic Phase Error Correction	Enables the dynamic sensor phase error correction.
4	Advanced Sensor Correction	Enables the advanced signal correction feature.
5 .. 31	Reserved	These bits are reserved for future use.

This property is stored to non-volatile memory. The default value is 0x0f (15) with means that the amplitude and phase error corrections are active.

Disabling the **Dynamic Amplitude and Phase Error Correction** might be useful for some special applications to achieve a higher position repeatability with the trade-off off a lower absolute position accuracy.

The **Advanced Sensor Correction** allows to compensate periodic sensor errors. The correction requires an additional calibration routine which must be performed once for every channel. This routine generates a compensation table for the sensor data which is applied to the position calculation if the `SA_CTL_SIGNAL_CORR_OPT_BIT_ASC` (bit 4) flag is set to one. See section 2.6.1 "Advanced Sensor Correction Calibration (calibration options `0x04` or `0x05`)" for the details on the calibration routine.



NOTICE
The advanced sensor correction needs a feature permission to be activated on the controller. See section 2.19 "Feature Permissions" for more information.

Example

```
// disable the dynamic amplitude and phase error correction for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_SIGNAL_CORRECTION_OPTIONS, 0
);
```

4.7 Referencing Properties

4.7.1 Referencing Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_REFERENCING_OPTIONS	0x0307005D	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:REFerencing:OPTions`

Description

This property specifies the find reference mode. It is used to define the behavior of the find reference routine when calling the `SA_CTL_Reference` function.

Note that the find reference sequence is also influenced by the Move Velocity and Move Acceleration properties (see there).

The value is a bit field containing independent flags. Please refer to section 2.7.1 "Reference Marks" for more information on the find reference sequence.

Undefined flags are reserved for future use. These flags should be set to zero.

The default mode is 0 (all flags cleared).

Example

```
SA_CTL_Result_t result;
int8_t channelIdx = 2;
// set find reference mode of channel 2 (start direction: backwards)
result = SA_CTL_SetProperty_i32(
    dHandle,
    channelIdx,
    SA_CTL_PKEY_REFERENCING_OPTIONS,
    SA_CTL_REF_OPT_BIT_START_DIR
);
if (result) {// handle error, abort}
// set velocity to 1mm/s
result = SA_CTL_SetProperty_i64(
    dHandle,channelIdx,SA_CTL_PKEY_MOVE_VELOCITY,1e9
);
if (result) {// handle error, abort}
// disable acceleration control
result = SA_CTL_SetProperty_i64(
    dHandle,channelIdx,SA_CTL_PKEY_MOVE_ACCELERATION,0
);
if (result == SA_CTL_ERROR_NONE) {
```

```
    // start searching for the reference with the previously
    // set parameters
    result = SA_CTL_Reference(dHandle, channelIdx, 0);
}
```

4.7.2 Distance To Reference Mark

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DISTANCE_TO_REF_MARK	0x030700A2	I64	Ch	R	X

ASCII Command: `[:PROPerty]:CHANnel#:REFerencing:DTRMark`

Description

This property holds the distance between the start of a referencing movement and the reference mark. Note that the position of the reference mark is not necessarily the position where the positioner comes to halt. The behavior depends on the Referencing Options. See section 2.6.2 "Referencing" for more information. The value is updated whenever a referencing sequence finished. The unit is pico meter (pm) for linear positioners and nano degree (n°) for rotatory positioners.

See Also

4.7.1 Referencing Options

4.7.3 Distance Code Inverted

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DIST_CODE_INVERTED	0x0307000E	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:REFerencing:DCINverted`

Description

This property is used to correct the absolute position calculation when referencing positioners with multiple reference marks. In rare cases the reference algorithm may produce faulty results due to a reference coding mismatch. The correct setting is determined by an automatic calibration routine, thus it is usually not necessary to manually modify this property.

This property is stored to non-volatile memory and need not be configured on every power-up.

See section 2.6.1 "Calibrating" for more information.

Valid Range

`SA_CTL_NON_INVERTED` (`0x00`), `SA_CTL_INVERTED` (`0x01`)

4.8 Tuning and Customizing Properties

4.8.1 Positioner Movement Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
`SA_CTL_PKEY_POS_MOVEMENT_TYPE`	`0x0309003F`	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:MTYPe`

Description

This property holds the positioner movement type. It may be used to determine the type of positioner (*linear*, *rotatory*, *goniometer* or *tip-tilt*) that is configured for the channel. This property has informational character only.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

`SA_CTL_POS_MOVEMENT_TYPE_LINEAR` (`0`),
`SA_CTL_POS_MOVEMENT_TYPE_ROTATORY` (`1`),
`SA_CTL_POS_MOVEMENT_TYPE_GONIOMETER` (`2`),
`SA_CTL_POS_MOVEMENT_TYPE_TIP_TILT` (`3`)

Example

```
SA_CTL_Result_t result;
int32_t type;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_MOVEMENT_TYPE, &type
);
if (result == SA_CTL_ERROR_NONE) {
    if (type == SA_CTL_POS_MOVEMENT_TYPE_GONIOMETER) {
        // goniometer type configured
    }
}
```

See Also

4.8.3 Positioner Base Unit, 4.8.4 Positioner Base Resolution

4.8.2 Positioner Is Custom Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_IS_CUSTOM_TYPE	0x03090041	I32	Ch	R	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:CUSTom`

Description

This property may be used to determine if the currently configured positioner type is a custom type.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Example

```
SA_CTL_Result_t result;
int32_t custom;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_IS_CUSTOM_TYPE, &custom
```

```
);
if (result == SA_CTL_ERROR_NONE) {
    if (custom) // custom positioner type configured
    else // predefined positioner type configured
}
```

See Also

4.4.6 Positioner Type

4.8.3 Positioner Base Unit

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_BASE_UNIT	0x03090042	I32	Ch	R(W)	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:BASE:UNIT

Description

This property holds the basic unit of the position values a channel uses. (e.g. meter, degree). Note that this property has informational character only. Setting it to a different value wont influence the position calculation.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

SA_CTL_UNIT_METER (0x00000002), SA_CTL_UNIT_DEGREE (0x00000003)

Example

```
SA_CTL_Result_t result;
int32_t unit;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_BASE_UNIT, &unit
);
```

```
if (result == SA_CTL_ERROR_NONE) {
    if (unit == SA_CTL_UNIT_METER)  // linear positioner type configured
    else // rotatory/goniometer positioner type configured
}
```

See Also

4.8.4 Positioner Base Resolution

4.8.4 Positioner Base Resolution

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_BASE_RESOLUTION	0x03090043	I32	Ch	R(W)	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:BASE:RESolution

Description

This property holds the basic resolution of the position value in powers of 10. It may be used to programmatically determine the interpretation of the position value of a channel. The resolution depends on the configured positioner type. (see Positioner Type) For example, a channel configured as linear positioner type has a base unit of *Meter* and a base resolution of −12. So a position value of 100 000 000 would correspond to 100 µm. Note that this property has informational character only. Setting it to a different value wont influence the position calculation.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

−12 . . . 0.

Example

```
SA_CTL_Result_t result;
int32_t resolution;
result = SA_CTL_GetProperty_i32(
```

```
    dHandle, 0, SA_CTL_PKEY_POS_BASE_RESOLUTION, &resolution
);
if (result == SA_CTL_ERROR_NONE) {
    // 'resolution' holds the base resolution of channel 0
}
```

See Also

4.8.3 Positioner Base Unit

4.8.5 Positioner Sensor Head Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_HEAD_TYPE	0x0309008E	I32	Ch	R(W)	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:HTYPe

Description

This property specifies the sensor head type. This property is only relevant if a SmarAct PicoScale interferometer is used as sensor module. The head type is set to the PicoScale when an adjustment sequence is started with the MCS2 hand control module.

For more information on head types refer to the PicoScale User Manual.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

C01, C02, C03, F01

See Also

4.8.17 Positioner Write Protection, 4.8.16 Save Positioner Type

4.8.6 Positioner Reference Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_REF_TYPE	0x03090048	I32	Ch	R(W)	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:RTYPe

Description

This property specifies the reference type of the positioner. The reference type is used by the SA_CTL_Reference function to determine the physical position. See section 2.7.1 "Reference Marks" for more information.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

SA_CTL_REF_TYPE_NONE (0),
SA_CTL_REF_TYPE_END_STOP (1),
SA_CTL_REF_TYPE_SINGLE_CODED (2),
SA_CTL_REF_TYPE_DISTANCE_CODED (3)

Example

```
SA_CTL_Result_t result;
int32_t type;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_REF_TYPE, &type
);
if (result == SA_CTL_ERROR_NONE) {
    if (type == SA_CTL_REF_TYPE_SINGLE_CODED) {
        // single coded reference type configured
    }
}
```

See Also

4.8.17 Positioner Write Protection, 4.8.16 Save Positioner Type

4.8.7 Positioner P Gain

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_P_GAIN	0x0309004B	I32	Ch	R(W)	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:GAIN:P

Description

This property specifies the proportional gain of the control-loop. Note that the resulting gain is also influenced by the Positioner PID Shift property.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

$0 \ldots 2 \times 10^9$.

Example

```
// set the P gain to 100 for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_P_GAIN, 100
);
```

See Also

4.8.8 Positioner I Gain, 4.8.9 Positioner D Gain, 4.8.10 Positioner PID Shift

4.8.8 Positioner I Gain

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_I_GAIN	0x0309004C	I32	Ch	R(W)	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:GAIN:I

Description

This property specifies the integral gain of the control-loop. The Positioner Anti Windup must be set to a non-zero value to activate the I gain of the control-loop. Note that the resulting gain is also influenced by the Positioner PID Shift property.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

$0 \ldots 2 \times 10^9$.

Example

```
// set the I gain to 0 for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_I_GAIN, 0
);
```

See Also

4.8.7 Positioner P Gain, 4.8.9 Positioner D Gain, 4.8.10 Positioner PID Shift, 4.8.11 Positioner Anti Windup

4.8.9 Positioner D Gain

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_D_GAIN	0x0309004D	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:GAIN:D`

Description

This property specifies the differential gain of the control-loop. Note that the resulting gain is also influenced by the Positioner PID Shift property.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

$0 \ldots 2 \times 10^9$.

Example

```
// set the D gain to 10 for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_P_GAIN, 10
);
```

See Also

4.8.7 Positioner P Gain, 4.8.8 Positioner I Gain, 4.8.10 Positioner PID Shift

4.8.10 Positioner PID Shift

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_PID_SHIFT	0x0309004E	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:GAIN:SHIFt`

Description

This property specifies a shift value for the PID controller output. Since PID parameters are configured as integer values they are right shifted internally to be able to set gains lower than one. It must be given in powers of 2.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

The default value is 10.

Valid Range

0 . . . 16.

Example

```
// set the PID shift to 10 (default) for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_PID_SHIFT, 10
);
```

See Also

4.8.7 Positioner P Gain, 4.8.8 Positioner I Gain, 4.8.9 Positioner D Gain

4.8.11 Positioner Anti Windup

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_ANTI_WINDUP	0x0309004F	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:AWINdup`

Description

This property specifies the anti windup limit for the integral gain of the control-loop. It has no meaning if the Positioner I Gain property is set to zero.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

The default value is 0.

Valid Range

$0 \ldots 2 \times 10^9$.

Example

```
// set the anti windup to default for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_ANTI_WINDUP, 0
);
```

See Also

4.8.7 Positioner P Gain, 4.8.8 Positioner I Gain, 4.8.9 Positioner D Gain, 4.8.10 Positioner PID Shift

4.8.12 Positioner ESD Distance Threshold

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_ESD_DIST_TH	0x03090050	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:ESDetection:DISTance`

Description

This property specifies the end stop detection distance threshold in pm or n°. This property in conjunction with the Positioner ESD Counter Threshold configure the end stop detection responsible to detect a physical end stop as well as a mechanical blockage of a positioner for closed-loop movements. An end stop condition leads to a stop of the channel.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Generally, there is no need to modify the end stop detection configuration. The configured Positioner Type defines appropriate values for all kinds of SmarAct positioners. Nonetheless it may be necessary to disable the end stop detection in some special cases. E.g. if an auxiliary input is used as feedback for the control-loop and the actual input value represents a set-point for the control-loop instead of a current position of the positioner.

The default value depends on the configured positioner type. The special value 0 disables the end stop detection.

CAUTION
Configuring inappropriate values or disabling the end stop detection prevents the channel from stopping the positioner in case of a mechanical blockage. The end stop detection configuration properties must be used with caution!

Valid Range

$0 \ldots 1 \times 10^9$.

Example

```
// set the end stop detection distance threshold to 1000000 for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_ESD_DIST_TH, 1000000
);
```

See Also

4.8.13 Positioner ESD Counter Threshold

4.8.13 Positioner ESD Counter Threshold

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_ESD_COUNTER_TH	0x03090051	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:ESDetection:COUNter`

Description

This property specifies the end stop detection counter threshold.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

$1 \ldots 4 \times 10^9$.

Example

```
// set the end stop detection counter value to 100000 for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_ESD_COUNTER_TH, 100000
);
```

See Also

4.8.12 Positioner ESD Distance Threshold

4.8.14 Positioner Target Reached Threshold

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_TARGET_REACHED_TH	0x03090052	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:THReshold:TREached`

Description

This property specifies the target reached threshold in pm or n°. A closed-loop movement is considered to be finished once the target position $\pm$ the target reached threshold is reached.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

Valid Range

$0 \ldots 1 \times 10^6$.

Example

```
// set the target reached threshold to 5nm for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_TARGET_REACHED_TH, 5000
);
```

See Also

4.8.15 Positioner Target Hold Threshold

4.8.15 Positioner Target Hold Threshold

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_TARGET_HOLD_TH	0x03090053	I32	Ch	R(W)	X

ASCII Command: `[:PROPerty]:CHANnel#:TUNing:THReshold:THOLd`

Description

This property specifies the target hold threshold in pm or n°. The hold threshold defines a dead zone around the control-loop input signal where the output does not change. This parameter is typically used in a system where the resolution of the sensor is significantly lower then the resolution of the actor. The dead zone then prevents oscillation or "hunting" of the control-loop.

Note that you must remove the write protection with the Positioner Write Protection property (see there) before being able to write this property.

This is a setting of the positioner type configuration. To make it persistent save the positioner type configuration to a custom slot. See section 2.5.1 "Custom Positioner Types" for more information.

The default value is 0.

Valid Range

$0 \ldots 1 \times 10^6$.

Example

```
// set the target hold threshold to 100nm for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_TARGET_HOLD_TH, 100000
);
```

See Also

4.8.14 Positioner Target Reached Threshold

4.8.16 Save Positioner Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_SAVE	0x0309000A	I32	Ch	W	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:SAVE

Description

This property is used to save a modified positioner type to a custom slot of a channel. Currently four custom slots per channel are available. Saving the positioner type makes the parameters persistent and implicit sets the Positioner Type to the given custom type.

Valid Range

SA_CTL_POSITIONER_TYPE_CUSTOM0 (250), SA_CTL_POSITIONER_TYPE_CUSTOM1 (251), SA_CTL_POSITIONER_TYPE_CUSTOM2 (252), SA_CTL_POSITIONER_TYPE_CUSTOM3 (253)

Example

```
// save a modified positioner type of channel 0 to custom slot 1
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_POS_SAVE, SA_CTL_POSITIONER_TYPE_CUSTOM0
);
```

See Also

4.8.17 Positioner Write Protection

4.8.17 Positioner Write Protection

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_POS_WRITE_PROTECTION	0x0309000D	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:TUNing:WPRotection

Description

This property is used to unlock the write access to the tuning parameters. A special key must be written to this property to unlock the write access to the tuning properties. Write any other value to this property to enable the protection again. Otherwise the write protection remains unlocked for the channel until the device is restarted. The write protection key is:
`SA_CTL_POS_WRITE_PROTECTION_KEY` (`0x534D4152`)

Example

```
// disable tuning parameter write protection of channel 0
result = SA_CTL_SetProperty_i32(
    dHandle,
    0,
    SA_CTL_PKEY_POS_WRITE_PROTECTION,
    SA_CTL_POS_WRITE_PROTECTION_KEY
);
// set tuning parameters like P gain, etc.
```

See Also

4.8.16 Save Positioner Type

4.9 Streaming Properties

4.9.1 Stream Base Rate

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
`SA_CTL_PKEY_STREAM_BASE_RATE`	`0x040F002C`	I32	Dev	RW	-

ASCII Command: `[:PROPerty]:DEVice:STReaming:BASerate`

Description

This property specifies the stream base rate in Hz for the trajectory streaming. Please refer to section 2.15 "Trajectory Streaming" for more information.

The default stream base rate is 1000 Hz.

Valid Range

10 . . . 1000 Hz

Example

```
// set the stream rate to 1 kHz
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_STREAM_BASE_RATE, 1000
);
```

See Also

4.9.2 Stream External Sync Rate, 4.13.1 Device Input Trigger Mode

4.9.2 Stream External Sync Rate

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_STREAM_EXT_SYNC_RATE	0x040F002D	I32	Dev	RW	-

ASCII Command: [:PROPerty]:DEVice:STReaming:SYNCrate

Description

This property specifies the external stream synchronization rate in Hz for the trajectory streaming. It may be used to synchronize the internal position streaming clock to an external clock signal. Note that the configured Stream Base Rate must be a whole-number multiple of the external sync rate.

The default value is 1.

Valid Range

1 . . . 1000 Hz

NOTICE
In order to use the external stream synchronization the device must be equipped with an trigger input connector.

Example

```
// configure external stream synchronization rate to 100Hz
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_STREAM_EXT_SYNC_RATE, 100
);
```

See Also

4.9.1 Stream Base Rate, 4.13.1 Device Input Trigger Mode

4.9.3 Stream Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_STREAM_OPTIONS	0x040F005D	I32	Dev	RW	-

ASCII Command: [:PROPerty]:DEVice:STReaming:OPTions

Description

This property specifies the stream's options. It is used to define the behavior of the stream before calling the SA_CTL_OpenStream function.

The value is a bit field containing independent flags. Please refer to the subsection 2.15.3 "Options" for more information. Undefined flags are unused but might get a meaning in future updates. Undefined flags should be set to zero. The default value is 0 (all flags cleared).

Example

```
// disable the target position interpolation for the trajectory streaming
result = SA_CTL_SetProperty_i32(
    dHandle,
    0,
    SA_CTL_PKEY_STREAM_OPTIONS,
    SA_CTL_STREAM_OPT_BIT_INTERPOLATION_DIS
);
```

See Also

4.9.1 Stream Base Rate

4.9.4 Stream Load Maximum

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_STREAM_LOAD_MAX	0x040F0301	I32	Dev	R	-

ASCII Command: N/A

Description

This property reports the maximum load generated by the current stream in percent. The property acts like a peak detector. The highest load level generated by the currently running stream is stored. When starting the stream the load value is reset to zero. Please refer to section 2.15 "Trajectory Streaming" for more information.

Valid Range

0 . . . 100 %

Example

```
SA_CTL_Result_t result;
int32_t maximumLoad;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_STREAM_LOAD_MAX, &maximumLoad
);
```

4.10 Diagnostic Properties

4.10.1 Channel Error

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CHANNEL_ERROR	0x0502007A	I32	Ch	R	X

ASCII Command: `[:PROPerty]:CHANnel#:ERRor`

Description

This property holds the last error of a channel. Generally, event notifications are used to inform about channel errors. (See section 2.6.7 "Movement Feedback" for more information.) However, if event notifications are not used in an application the Channel State bit `SA_CTL_CH_STATE_BIT_MOVEMENT_FAILED` can be monitored to detect channel errors. This property may be read then to determine the reason of the error.

Note that the channel error is reset to `SA_CTL_ERROR_NONE` after reading this property.

Example

```
SA_CTL_Result_t result;
int32_t chError;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_CHANNEL_ERROR, &chError
);
if (result == SA_CTL_ERROR_NONE) {
    // 'chError' holds the last error code of channel 0
}
```

See Also

4.4.9 Channel State

4.10.2 Channel Temperature

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CHANNEL_TEMPERATURE	0x05020034	I32	Ch	R	X

ASCII Command: `[:PROPerty]:CHANnel#:TEMPerature`

Description

This property holds the amplifier temperature in °C. The temperature is measured near the channels driver amplifier.

Example

```
SA_CTL_Result_t result;
int32_t chTemp;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_CHANNEL_TEMPERATURE, &chTemp
);
if (result == SA_CTL_ERROR_NONE) {
    // 'chTemp' holds the temperature of the amplifier of channel 0
}
```

See Also

4.10.3 Bus Module Temperature

4.10.3 Bus Module Temperature

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_BUS_MODULE_TEMPERATURE	0x05030034	I32	Mod	R	X

ASCII Command: `[:PROPerty]:MODule#:TEMPerature`

Description

This property holds the temperature of a bus module in °C.

Example

```
SA_CTL_Result_t result;
int32_t modTemp;
result = SA_CTL_GetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_BUS_MODULE_TEMPERATURE, &modTemp
);
if (result == SA_CTL_ERROR_NONE) {
    // 'modTemp' holds the temperature of the driver module 0
}
```

See Also

4.10.2 Channel Temperature

4.11 Auxiliary Properties

4.11.1 Aux Positioner Type

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_POSITIONER_TYPE	0x0802003C	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:AUXiliary:PTYPe

Description

This property is used to tell the channel which set of control-loop parameters (PID gains, etc.) is used when an auxiliary input is configured as input for the control-loop. More precisely, if the Control Loop Input property is set to SA_CTL_CONTROL_LOOP_INPUT_AUX_IN the auxiliary positioner type parameters are implicitly configured, otherwise the regular positioner type parameters are used. This way it is possible to switch between two control modes without manually changing all individual parameters. Typically a custom positioner type slot will be used here to define the necessary parameters.

Please refer to section 2.16.5 "Using Analog Inputs as Control-Loop Feedback" for more information on using auxiliary inputs.

This property is stored to non-volatile memory and need not be configured on every power-up.

Valid Range

SA_CTL_POSITIONER_TYPE_CUSTOM0 (250),
SA_CTL_POSITIONER_TYPE_CUSTOM1 (251),
SA_CTL_POSITIONER_TYPE_CUSTOM2 (252),
SA_CTL_POSITIONER_TYPE_CUSTOM3 (253)

Example

```
// select the 'CUSTOM0' positioner type (type code 250) for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_AUX_POSITIONER_TYPE, 250
);
```

See Also

4.4.6 Positioner Type

4.11.2 Aux Positioner Type Name

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_POSITIONER_TYPE_NAME	0x0802003D	String	Ch	R	-

ASCII Command: [:PROPerty]:CHANnel#:AUXiliary:PTName

Description

This property holds a descriptive name of the configured auxiliary positioner type. The positioner type name is a null terminated string. Note that the name is read-only.

Example

```
SA_CTL_Result_t result;
char name[SA_CTL_STRING_MAX_LENGTH];
size_t ioStringSize = sizeof(name);
result = SA_CTL_GetProperty_s(
    dHandle, 0, SA_CTL_PKEY_AUX_POSITIONER_TYPE_NAME, name, &ioStringSize
);
if (result == SA_CTL_ERROR_NONE) {
    // 'name' holds the name of the configured auxiliary positioner type
}
```

See Also

4.11.1 Aux Positioner Type, 4.4.6 Positioner Type

4.11.3 Aux Input Select

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_INPUT_SELECT	0x08020018	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:AUXiliary:ISELect

Description

This property selects the auxiliary input component. Note that the Aux I/O Module Input Index property must be configured too to select a specific analog input.

Note that the additional sensor module inputs are not available on all sensor module types. Please refer to section 2.16 "Auxiliary Inputs and Outputs" for more information on using auxiliary inputs.

This property is stored to non-volatile memory and need not be configured on every power-up. The default value is `SA_CTL_AUX_INPUT_SELECT_IO_MODULE` (`0`).

Valid Range

`SA_CTL_AUX_INPUT_SELECT_IO_MODULE` (`0`),
`SA_CTL_AUX_INPUT_SELECT_SENSOR_MODULE` (`1`)

Example

```
// set the auxiliary input selection to 'I/O module' for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_AUX_INPUT_SELECT,
    SA_CTL_AUX_INPUT_SELECT_IO_MODULE
);
```

See Also

4.11.4 Aux I/O Module Input Index, 4.11.5 Aux Direction Inversion, 4.4.28 Control Loop Input Aux Value, 4.4.4 Control Loop Input

4.11.4 Aux I/O Module Input Index

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_IO_MODULE_INPUT_INDEX	0x081100AA	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:AUXiliary:IOModule:INPut:INDex`

Description

This property specifies which input of an analog I/O module is used as input for the auxiliary control-loop input.

The I/O module has a total number of six analog inputs which are mapped in groups of two to the channels of the corresponding driver module. The input index refers to the analog inputs assigned to a specific channel as follows:

Input Index	Channel Index	Analog Input
0	0 (3) (6)	AIN-1
0	1 (4) (7)	AIN-2
0	2 (5) (8)	AIN-3
1	0 (3) (6)	AIN-4
1	1 (4) (7)	AIN-5
1	2 (5) (8)	AIN-6

Note that input indexes refer to a module (start with zero for each module) while the channel indexes refer to the entire device. Channel indexes in brackets refer to a second respectively third module of the device.

Please refer to section 2.16 "Auxiliary Inputs and Outputs" for more information on using auxiliary inputs. See the MCS2 User Manual for the pin assignment of the I/O module connector.

This property is stored to non-volatile memory and need not be configured on every power-up. The default input index is 0.

Valid Range

0 . . . 1

Example

```
// set the auxiliary I/O module input index to 0 for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_AUX_IO_MODULE_INPUT_INDEX, 0
);
```

See Also

4.11.3 Aux Input Select, 4.11.5 Aux Direction Inversion, 4.11.6 Aux I/O Module Input0 / Input1 Value

4.11.5 Aux Direction Inversion

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_DIRECTION_INVERSION	0x0809000E	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:AUXiliary:DINVersion

Description

This property specifies the feedback direction sense for the control-loop in case an auxiliary input is used as input for the control-loop. The direction sense of the feedback must match the direction sense of the control-loop output. Otherwise a runaway condition may occur when commanding a closed-loop movement. The end stop detection (if not disabled) will typically abort the movement in that case. While the direction sense is determined automatically by the calibration routine when using the position as feedback signal, this setting must be defined manually using this property when using an auxiliary input. This property has no meaning if the Control Loop Input is not configured to auxiliary input.

Please refer to section 2.16.5 "Using Analog Inputs as Control-Loop Feedback" for more information on using auxiliary inputs.

This property is stored to non-volatile memory and need not be configured on every power-up. The default is SA_CTL_NON_INVERTED (0x00).

Valid Range

SA_CTL_NON_INVERTED (0x00), SA_CTL_INVERTED (0x01)

Example

```
// set the auxiliary direction inversion to 'inverted' for channel 0
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_AUX_DIRECTION_INVERSION, SA_CTL_INVERTED
);
```

See Also

4.11.3 Aux Input Select, 4.11.6 Aux I/O Module Input0 / Input1 Value, 4.4.4 Control Loop Input

4.11.6 Aux I/O Module Input0 / Input1 Value

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_IO_MODULE_INPUT0_VALUE	0x08110000	I64	Ch	R	X
SA_CTL_PKEY_AUX_IO_MODULE_INPUT1_VALUE	0x08110001	I64	Ch	R	X

ASCII Command: `[:PROPerty]:CHANnel#:AUXiliary:IOModule:INPut:VALue#`

Description

These properties hold the input values of the analog inputs of an analog I/O module. Note that an error is returned if no I/O module is available.

Note further that the interpretation of the value depends on the configured I/O Module Analog Input Range of the I/O module. Please refer to section 2.16 "Auxiliary Inputs and Outputs" for more information on using auxiliary inputs.

Example

```
SA_CTL_Result_t result;
int64_t inputVal;
result = SA_CTL_GetProperty_i64(
    dHandle, 0, SA_CTL_PKEY_AUX_IO_MODULE_INPUT0_VALUE, &inputVal
);
if (result == SA_CTL_ERROR_NONE) {
    // `inputVal` holds the current input value of the first
    // I/O module input of channel 0
}
```

See Also

4.11.3 Aux Input Select, 4.4.28 Control Loop Input Aux Value, 4.4.27 Control Loop Input Sensor Value, 4.11.6 Aux I/O Module Input0 / Input1 Value, 4.4.4 Control Loop Input

4.11.7 Aux Digital Input Value

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_DIGITAL_INPUT_VALUE	0x080300AD	I32	Mod	R	X

ASCII Command: `[:PROPerty]:MODule#:AUXiliary:DINPut[:VALue]`

Description

This property holds a bit mask that represents the input levels of the general purpose digital inputs of an I/O module.

Please refer to section 2.16 "Auxiliary Inputs and Outputs" for more information.

Example

```
SA_CTL_Result_t result;
// read the digital inputs
int32_t input;
result = SA_CTL_GetProperty_i32(dHandle,0,
    SA_CTL_PKEY_AUX_DIGITAL_INPUT_VALUE, &input
);
if (result == SA_CTL_ERROR_NONE) {
    // 'input' holds the value of the digital inputs
}
```

See Also

4.11.8 Aux Digital Output Value / Set / Clear

4.11.8 Aux Digital Output Value / Set / Clear

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_VALUE	0x080300AE	I32	Mod	RW	X
SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_SET	0x080300B0	I32	Mod	W	X
SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_CLEAR	0x080300B1	I32	Mod	W	X

ASCII Command: `[:PROPerty]:MODule#:AUXiliary:DOUTput[:VALue]`
ASCII Command: `[:PROPerty]:MODule#:AUXiliary:DOUTput:SET`
ASCII Command: `[:PROPerty]:MODule#:AUXiliary:DOUTput:CLEar`

Description

These properties hold bit masks that may be used to modify the general purpose digital outputs of an I/O module. Note that the digital output driver circuit is disabled by default and must be enabled by setting the SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUT_ENABLED bit of the I/O Module Options property.

Please refer to section 2.16 "Auxiliary Inputs and Outputs" for more information.

Example

```
SA_CTL_Result_t result;
// set all digital output of the I/O module to a specific value
// DOUT-4 | DOUT-3 | DOUT-2 | DOUT-1 |
// L(0)   | H(1)   | L(0)   | H(1)   |
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_DIGITAL_OUTPUT_VALUE, 0x00000005
);
```

See Also

4.11.7 Aux Digital Input Value, 4.12.1 I/O Module Options

4.11.9 Aux Analog Output Value0 / Value1

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUX_ANALOG_OUTPUT_VALUE0	0x08030000	I32	Mod	RW	X
SA_CTL_PKEY_AUX_ANALOG_OUTPUT_VALUE1	0x08030001	I32	Mod	RW	X

ASCII Command: [:PROPerty]:MODule#:AUXiliary:AOUTput:VALue#

Description

These properties specify the output values of the analog outputs of an I/O module. Note that the analog output driver circuit is in a high-impedance state by default and must be enabled by setting the SA_CTL_IO_MODULE_OPT_BIT_ANALOG_OUTPUT_ENABLED bit of the I/O Module Options property.

The output values are given as signed 16-bit values from −32 768 to 32 767, where −32 768 corresponds to −10 V and 32 767 to 10 V output voltage.

The default value is 0 which corresponds to an output voltage of 0 V.

Valid Range

−32 768 . . . 32 767

Example

```
SA_CTL_Result_t result;
// set the output value of analog output0 (AOUT-1) to zero
// which corresponds to 0V
result = SA_CTL_SetProperty_i32(dHandle,0,
  SA_CTL_PKEY_AUX_ANALOG_OUTPUT_VALUE0, 0
);
```

See Also

4.11.6 Aux I/O Module Input0 / Input1 Value, 4.12.1 I/O Module Options

4.12 I/O Module Properties

4.12.1 I/O Module Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_IO_MODULE_OPTIONS	0x0603005D	I32	Mod	RW	X

ASCII Command: [:PROPerty]:MODule#:IOModule:OPTions

Description

This property specifies the I/O module options. The value is a bit field containing independent flags with the following meaning:

Bit	Name	Short Desrciption
0	SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUT_ENABLED	Enables or disables the digital output driver circuit on the I/O module.
1	SA_CTL_IO_MODULE_OPT_BIT_EVENTS_ENABLED	Enables or disables the event notification for the digital inputs of an I/O module.
2	SA_CTL_IO_MODULE_OPT_BIT_ANALOG_OUTPUT_ENABLED	Enables or disables the analog output driver circuit on the I/O module.
3 .. 31	Reserved	These bits are reserved for future use.

All options are disabled by default, which means that all digital and analog outputs are in a high-impedance state and the digital input events are disabled.

NOTICE

Note that the *events enabled* bit refers to the general purpose digital inputs of the I/O module and **not** to the digital device trigger input. See section 2.17 "Input Trigger" for the event notification configuration of the device input trigger.

Note that the I/O Module Voltage property should be set first to define the voltage level of the digital outputs.

Example

```
// enable the digital and analog output driver circuit of the I/O module
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_IO_MODULE_OPTIONS,
    (SA_CTL_IO_MODULE_OPT_BIT_DIGITAL_OUTPUT_ENABLED |
     SA_CTL_IO_MODULE_OPT_BIT_ANALOG_OUTPUT_ENABLED)
);
```

See Also

4.12.2 I/O Module Voltage

4.12.2 I/O Module Voltage

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_IO_MODULE_VOLTAGE	0x06030031	I32	Mod	RW	X

ASCII Command: [:PROPerty]:MODule#:IOModule:VOLTage

Description

This property specifies the I/O module output voltage for the digital outputs. The output voltage should be set before enabling the outputs of the I/O module. Note that the voltage setting is global for all digital output channels of the I/O module.

The default value is SA_CTL_IO_MODULE_VOLTAGE_3V3 (0).

Valid Range

SA_CTL_IO_MODULE_VOLTAGE_3V3 (0),
SA_CTL_IO_MODULE_VOLTAGE_5V (1)

Example

```
// set the output driver voltage level to 5V
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_IO_MODULE_VOLTAGE,
    SA_CTL_IO_MODULE_VOLTAGE_5V
);
```

See Also

4.12.1 I/O Module Options

4.12.3 I/O Module Analog Input Range

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_IO_MODULE_ANALOG_INPUT_RANGE	0x060300A0	I32	Mod	RW	X

ASCII Command: [:PROPerty]:MODule#:IOModule:AINPut:RANGe

Description

This property specifies the I/O module analog input range. This setting configures the analog gain settings of the ADCs of the I/O module. The inputs allow bipolar as well as unipolar operation. To achieve the best performance of the ADC it is recommended to always use the lowest full range setting that fits the desired analog input range.

Note that the range setting does not influence the digital representation of the input value. The signed value of 2^{17} corresponds to a bipolar full range input of 10.24 V. This means that e.g. an analog voltage of 2.56 V always returns a digital value of 32767 regardless of the actual range setting. The advantage of this representation is that e.g. configured PID gains or threshold limits must not be adjusted after changing the input range while the best matching analog gain is used for the analog to digital conversion. The following table summarizes the digital representations of the analog input voltage and their maximum values for the different gain settings:

Analog Voltage	Bipol. ±10V	Bipol. ±5V	Bipol. ±2.5V	Unipol. 10V	Unipol. 5V
+10.24V	131071	65535	32767	131071	65535
+5.12V	65535	65535	32767	65535	65535
+2.56V	32767	32767	32767	32767	32767
0V	0	0	0	0	0
-2.56V	-32768	-32768	-32768	0	0
-5.12V	-65536	-65536	-32768	0	0
-10.24V	-131072	-65536	-32768	0	0

Note that the input range setting is global for all analog inputs of the I/O module.

This property is stored to non-volatile memory and need not be configured on every power-up. The default value is `SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_BI_10V (0)`.

Valid Range

`SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_BI_10V (0)`,
`SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_BI_5V (1)`,
`SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_BI_2_5V (2)`,
`SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_UNI_10V (3)`,
`SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_UNI_5V (4)`

Example

```
// set the analog input range to +/-5V
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_IO_MODULE_ANALOG_INPUT_RANGE,
    SA_CTL_IO_MODULE_ANALOG_INPUT_RANGE_BI_5V
);
```

See Also

4.12.1 I/O Module Options, 4.11.6 Aux I/O Module Input0 / Input1 Value

4.13 Input Trigger Properties

4.13.1 Device Input Trigger Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DEV_INPUT_TRIG_MODE	0x060D0087	I32	Dev	RW	-

ASCII Command: [:PROPerty]:DEVice:TRIGger:INPut:MODE

Description

This property specifies the input trigger mode of the device. The input trigger may be used to synchronize the device to external events. If no I/O module is available this property returns a SA_CTL_ERROR_NO_IOM_PRESENT error. Please refer to section 2.17 "Input Trigger" for more information.

The default value is SA_CTL_DEV_INPUT_TRIG_MODE_DISABLED (0).

Valid Range

SA_CTL_DEV_INPUT_TRIG_MODE_DISABLED (0),
SA_CTL_DEV_INPUT_TRIG_MODE_EMERGENCY_STOP (1),
SA_CTL_DEV_INPUT_TRIG_MODE_STREAM (2),
SA_CTL_DEV_INPUT_TRIG_MODE_CMD_GROUP (3)
SA_CTL_DEV_INPUT_TRIG_MODE_EVENT (4)

Example

```
// set input trigger mode to external stream sync
result = SA_CTL_SetProperty_i32(
    dHandle,
    0,
    SA_CTL_PKEY_DEV_INPUT_TRIG_MODE,
    SA_CTL_DEV_INPUT_TRIG_MODE_STREAM
);
```

See Also

4.13.2 Device Input Trigger Condition, 4.9.1 Stream Base Rate, 4.9.2 Stream External Sync Rate

4.13.2 Device Input Trigger Condition

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_DEV_INPUT_TRIG_CONDITION	0x060D005A	I32	Dev	RW	-

ASCII Command: `[:PROPerty]:DEVice:TRIGger:INPut:CONDition`

Description

This property defines the active edge for the input trigger signal.

The default value is `SA_CTL_TRIGGER_CONDITION_RISING` (0).

Valid Range

`SA_CTL_TRIGGER_CONDITION_RISING` (0), `SA_CTL_TRIGGER_CONDITION_FALLING` (1)

Example

```
// set input trigger condition to "rising"
result = SA_CTL_SetProperty_i32(
    dHandle,
    0,
    SA_CTL_PKEY_DEV_INPUT_TRIG_CONDITION,
    SA_CTL_TRIGGER_CONDITION_RISING
);
```

See Also

4.13.1 Device Input Trigger Mode, 4.9.1 Stream Base Rate, 4.9.2 Stream External Sync Rate

4.14 Output Trigger Properties

4.14.1 Channel Output Trigger Mode

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_OUTPUT_TRIG_MODE	0x060E0087	I32	Ch	RW	X

ASCII Command: [:PROPerty]:CHANnel#:TRIGger:OUTPut:MODE

Description

This property specifies the output trigger mode of a channel. Note that further configuration of the output trigger should be done before it is enabled. If no I/O module is available this property returns a SA_CTL_ERROR_NO_IOM_PRESENT error.

Note for the position compare mode: if the Channel Position Compare Limit Max is set to a lower value than the Channel Position Compare Limit Min then this misconfiguration is indicated by a returned SA_CTL_ERROR_INVALID_CONFIGURATION error.

Please refer to section 2.18 "Output Trigger" for more information.

The default value is SA_CTL_CH_OUTPUT_TRIG_MODE_CONSTANT (0).

Valid Range

SA_CTL_CH_OUTPUT_TRIG_MODE_CONSTANT (0),
SA_CTL_CH_OUTPUT_TRIG_MODE_POSITION_COMPARE (1),
SA_CTL_CH_OUTPUT_TRIG_MODE_TARGET_REACHED (2),
SA_CTL_CH_OUTPUT_TRIG_MODE_ACTIVELY_MOVING (3)

Example

```
// set output trigger mode for channel 1
result = SA_CTL_SetProperty_i32(
    dHandle,
    1,
    SA_CTL_PKEY_CH_OUTPUT_TRIG_MODE,
    SA_CTL_CH_OUTPUT_TRIG_MODE_POSITION_COMPARE
);
```

See Also

4.14.4 Channel Position Compare Start Threshold, 4.14.5 Channel Position Compare Increment, 4.14.6 Channel Position Compare Direction, 4.14.2 Channel Output Trigger Polarity, 4.14.3 Channel Output Trigger Pulse Width

4.14.2 Channel Output Trigger Polarity

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY	0x060E005B	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:OUTPut:POLarity`

Description

This property defines the polarity of the output trigger signal. If set to *active high* then the idle level is low and a high pulse is generated when the trigger occurs. If set to *active low* then the idle level is high and a low pulse is generated when the trigger occurs.

The default polarity is `SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH` (1).

Valid Range

`SA_CTL_TRIGGER_POLARITY_ACTIVE_LOW` (0),
`SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH` (1)

Example

```
// set output trigger polarity for channel 1 to 'active high'
result = SA_CTL_SetProperty_i32(
    dHandle,
    1,
    SA_CTL_PKEY_CH_OUTPUT_TRIG_POLARITY,
    SA_CTL_TRIGGER_POLARITY_ACTIVE_HIGH
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.4 Channel Position Compare Start Threshold, 4.14.5 Channel Position Compare Increment, 4.14.6 Channel Position Compare Direction, 4.14.3 Channel Output Trigger Pulse Width

4.14.3 Channel Output Trigger Pulse Width

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_OUTPUT_TRIG_PULSE_WIDTH	0x060E005C	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:OUTPut:PWIDth`

Description

This property specifies the pulse width of the trigger output pulse in ns.

Note that the configured pulse width includes the duration of the pulse as well as the duration of the pause. E.g. when setting the Channel Output Trigger Pulse Width to 1000 ns pulses with 500 ns high level and 500 ns low level will be generated.

The default pulse width is 1000 ns.

Valid Range

100 ns . . . 4×10^9 ns

Example

```
// set output trigger pulse width for channel 1 to 1us
result = SA_CTL_SetProperty_i32(
    dHandle, 1, SA_CTL_PKEY_CH_OUTPUT_TRIG_PULSE_WIDTH, 1000
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.4 Channel Position Compare Start Threshold, 4.14.5 Channel Position Compare Increment, 4.14.6 Channel Position Compare Direction, 4.14.2 Channel Output Trigger Polarity

4.14.4 Channel Position Compare Start Threshold

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_POS_COMP_START_THRESHOLD	0x060E0058	I64	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:PCOMpare:THReshold[:STARt]`

Description

This property defines the start threshold value in pm or n° for the position compare output trigger. As soon as the position passes this threshold in the configured direction (see Channel Position Compare Direction) an output pulse is generated. Additionally the threshold is incremented by the value of the Channel Position Compare Increment to define the next trigger threshold. Please refer to section 2.18 "Output Trigger" for more information.

The default value is 1×10^9.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$ pm or n°.

Example

```
// set output trigger start threshold for channel 1 to 1mm
result = SA_CTL_SetProperty_i64(
    dHandle, 1, SA_CTL_PKEY_CH_POS_COMP_START_THRESHOLD, 1e9
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.5 Channel Position Compare Increment, 4.14.6 Channel Position Compare Direction, 4.14.2 Channel Output Trigger Polarity, 4.14.3 Channel Output Trigger Pulse Width

4.14.5 Channel Position Compare Increment

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_POS_COMP_INCREMENT	0x060E0059	I64	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:PCOMpare:INCRement`

Description

This property defines the position compare output trigger increment in pm or n°. Please refer to section 2.18 "Output Trigger" for more information.

The default value is 1×10^9.

Valid Range

$1 \ldots 1 \times 10^{12}$ pm or n°.

Example

```
// set position compare increment for channel 1 to 100um
result = SA_CTL_SetProperty_i64(
    dHandle, 1, SA_CTL_PKEY_CH_POS_COMP_INCREMENT, 100e6
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.4 Channel Position Compare Start Threshold, 4.14.6 Channel Position Compare Direction, 4.14.2 Channel Output Trigger Polarity, 4.14.3 Channel Output Trigger Pulse Width

4.14.6 Channel Position Compare Direction

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_POS_COMP_DIRECTION	0x060E0026	I32	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:PCOMpare:DIRection`

Description

This property defines how the position value and the configured trigger threshold are compared against each other.

The following trigger conditions are available:

Condition	Name	Short Description
0x00	SA_CTL_FORWARD_DIRECTION	The trigger pulse is output when the position value passes the threshold from below.
0x01	SA_CTL_BACKWARD_DIRECTION	The trigger pulse is output when the position value passes the threshold from above.
0x02	SA_CTL_EITHER_DIRECTION	The trigger pulse is output when the position value passes the threshold from below or above.

Please refer to section 2.18 "Output Trigger" for more information.

The default direction is SA_CTL_FORWARD_DIRECTION (0x00).

Example

```
// set output trigger condition for channel 1 to forward
result = SA_CTL_SetProperty_i32(
    dHandle,
    1,
    SA_CTL_PKEY_CH_POS_COMP_DIRECTION,
    SA_CTL_FORWARD_DIRECTION
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.4 Channel Position Compare Start Threshold, 4.14.5 Channel Position Compare Increment, 4.14.2 Channel Output Trigger Polarity, 4.14.3 Channel Output Trigger Pulse Width 4.14.7 Channel Position Compare Limit Min, 4.14.8 Channel Position Compare Limit Max

4.14.7 Channel Position Compare Limit Min

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_POS_COMP_LIMIT_MIN	0x060E0020	I64	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:PCOMpare:LMIN`

Description

This property defines the lower limit for the position compare output trigger in pm or n°. The limits act as an additional gate for the generation of output pulses. Output pulses are only generated when the current position lies between the configured minimum and maximum limits. Note that the maximum limit must be configured to a higher value than the minimum limit for the limit checks to be active. If both limits are set to the same value the checks are disabled and output pulses are generated according to the configured start threshold, increment and direction. Please refer to section 2.18 "Output Trigger" for more information.

The default value is 0.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$ pm or n°.

Example

```
// set position compare lower limit for channel 1 to 1mm
result = SA_CTL_SetProperty_i64(
    dHandle, 1, SA_CTL_PKEY_CH_POS_COMP_LIMIT_MIN, 1e9
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.4 Channel Position Compare Start Threshold, 4.14.5 Channel Position Compare Increment, 4.14.6 Channel Position Compare Direction, 4.14.2 Channel Output Trigger Polarity, 4.14.3 Channel Output Trigger Pulse Width, 4.14.8 Channel Position Compare Limit Max

4.14.8 Channel Position Compare Limit Max

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_CH_POS_COMP_LIMIT_MAX	0x060E0021	I64	Ch	RW	X

ASCII Command: `[:PROPerty]:CHANnel#:TRIGger:PCOMpare:LMAX`

Description

This property defines the upper limit for the position compare output trigger in pm or n°. The limits act as an additional gate for the generation of output pulses. Output pulses are only generated when the current position lies between the configured minimum and maximum limits. Note that the maximum limit must be configured to a higher value than the minimum limit for the limit checks to be active. If both limits are set to the same value the checks are disabled and output pulses are generated according to the configured start threshold, increment and direction. Please refer to section 2.18 "Output Trigger" for more information.

The default value is 0.

Valid Range

$-100 \times 10^{12} \ldots 100 \times 10^{12}$ pm or n°.

Example

```
// set position compare upper limit for channel 1 to 2mm
result = SA_CTL_SetProperty_i64(
    dHandle, 1, SA_CTL_PKEY_CH_POS_COMP_LIMIT_MAX, 2e9
);
```

See Also

4.14.1 Channel Output Trigger Mode, 4.14.4 Channel Position Compare Start Threshold, 4.14.5 Channel Position Compare Increment, 4.14.6 Channel Position Compare Direction, 4.14.2 Channel Output Trigger Polarity, 4.14.3 Channel Output Trigger Pulse Width, 4.14.7 Channel Position Compare Limit Min

4.15 Hand Control Module Properties

4.15.1 Hand Control Module Lock Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_HM_LOCK_OPTIONS	0x020C0083	I32	Dev	RW	-

ASCII Command: `[:PROPerty]:DEVice:HMODule:LOPTions[:CURRent]`

Description

This property defines the different possible lock states of an attached hand control module. The value is a bit field containing independent flags with the following meaning:

Table 4.2 – Hand Control Module Lock Options Bits

Bit	Name	Short Description
0	SA_CTL_HM1_LOCK_OPT_BIT_GLOBAL	Fully disables control over the hand controller.
1	SA_CTL_HM1_LOCK_OPT_BIT_CONTROL	Disables the control inputs (Encoder, Joystick, etc.).
4	SA_CTL_HM1_LOCK_OPT_BIT_CHANNEL_MENU	Hides the *Channel Settings* menu.
5	SA_CTL_HM1_LOCK_OPT_BIT_GROUP_MENU	Hides the *Group Settings* menu.
6	SA_CTL_HM1_LOCK_OPT_BIT_SETTINGS_MENU	Hides the General *Settings* menu.
7	SA_CTL_HM1_LOCK_OPT_BIT_LOAD_CFG_MENU	Hides the *Load Config* menu.
8	SA_CTL_HM1_LOCK_OPT_BIT_SAVE_CFG_MENU	Hides the *Save Config* menu.
9	SA_CTL_HM1_LOCK_OPT_BIT_CTRL_MODE_PARAM_MENU	Hides the generic control mode parameter menu.
12	SA_CTL_HM1_LOCK_OPT_BIT_CHANNEL_NAME	Hides the *Set Channel Name* menu entry.
13	SA_CTL_HM1_LOCK_OPT_BIT_POS_TYPE	Hides the *Positioner Type* menu entry.
14	SA_CTL_HM1_LOCK_OPT_BIT_SAFE_DIR	Hides the *Safe Direction* menu entry.
15	SA_CTL_HM1_LOCK_OPT_BIT_CALIBRATE	Hides the *Sensor Calibration* menu.
16	SA_CTL_HM1_LOCK_OPT_BIT_REFERENCE	Hides the *Find Reference* menu entry.
17	SA_CTL_HM1_LOCK_OPT_BIT_SET_POSITION	Hides the *Set Zero Position* menu entry.
18	SA_CTL_HM1_LOCK_OPT_BIT_MAX_CLF	Hides the *Max Closed-Loop Frequency* menu entry.
19	SA_CTL_HM1_LOCK_OPT_BIT_POWER_MODE	Hides the *Sensor Power Mode* menu entry.
20	SA_CTL_HM1_LOCK_OPT_BIT_ACTUATOR_MODE	Hides the *Actuator Mode* menu entry.

Note that this property is volatile. In order to alter the lock bits across sessions use the Hand Control Module Default Lock Options property instead.

Example

```
// disable control inputs for the hand control module
result = SA_CTL_SetProperty_i32(
    dHandle,0,SA_CTL_PKEY_HM_LOCK_OPTIONS,SA_CTL_HM1_LOCK_OPT_BIT_CONTROL
);
```

See Also

4.15.2 Hand Control Module Default Lock Options

4.15.2 Hand Control Module Default Lock Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_HM_DEFAULT_LOCK_OPTIONS	0x020C0084	I32	Dev	RW	-

ASCII Command: `[:PROPerty]:DEVice:HMODule:LOPTions:DEFault`

Description

This property specifies the default lock state of the hand control module at startup. It is the non-volatile version of the Hand Control Module Lock Options property. See table 4.2 for a description of the bit field.

Example

```
// hide channel and group menu by default
int32_t defaultLockState = (SA_CTL_HM1_LOCK_OPT_BIT_CHANNEL_MENU |
                            SA_CTL_HM1_LOCK_OPT_BIT_GROUP_MENU);
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_HM_DEFAULT_LOCK_OPTIONS, defaultLockState
);
```

See Also

4.15.1 Hand Control Module Lock Options

4.16 API Properties

4.16.1 Event Notification Options

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_EVENT_NOTIFICATION_OPTIONS	0xF010005D	I32	API	RW	-

ASCII Command: N/A

Description

This property may be used to configure the event notifications of the API.

The value is a bit field containing independent flags. Undefined flags are reserved for future use. These flags should be set to zero. The default value is 0 (all API events disabled).

NOTICE

Although this property is a setting of the API, an active connection to a device is still required. The setting applies to every individual device connection independently. Closing the connection to a device resets the setting to its default.

Table 4.3 – Event Notification Option Bits

Bit	Name	Short Description
0	SA_CTL_EVT_OPT_BIT_REQUEST_READY_ENABLED	Enable generation of request ready events.

Please refer to section 2.3.5 "Request Ready Notification" for more information on the request ready notifications.

Changing this property affects only new requests sent out after changing this property, not requests that were sent out before but have not received an answer yet.

Example

```
// enable the request ready events of the API
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_EVENT_NOTIFICATION_OPTIONS,
    SA_CTL_EVT_OPT_BIT_REQUEST_READY_ENABLED
);
```

See Also

2.4 Event Notifications, 2.3.5 Request Ready Notification, 5.2.20 Request Ready Event

4.16.2 Auto Reconnect

Definition

C-Definition	Code	Type	Idx	Access	CG[1]
SA_CTL_PKEY_AUTO_RECONNECT	0xF01000A1	I32	API	RW	-

ASCII Command: N/A

Description

This property configures the automatic reconnect feature of the API. In the default configuration the reconnect feature is disabled. When enabled the API detects lost connections and tries to reconnect to the device. Note that during the reconnect all device requests functions block until the reconnect is finished.

NOTICE

Although this property is a setting of the API, an active connection to a device is still required. The setting applies to every individual device connection independently. Closing the connection to a device resets the setting to its default.

Valid Range

SA_CTL_ENABLED (0x01), SA_CTL_DISABLED (0x00)

Example

```
// enable automatic reconnect
result = SA_CTL_SetProperty_i32(
    dHandle, 0, SA_CTL_PKEY_AUTO_RECONNECT, SA_CTL_ENABLED);
```

5 *EVENT REFERENCE*

5.1 Event Summary

An event always carries a 32-bit parameter. The meaning of this parameter depends on the event. The last column in the following table indicates the usage of the parameter.

Table 5.1 – Event Summary

Event	Code	Index	Parameter	Page
None	0x0000	N/A	N/A	231
Movement Finished	0x0001	Ch	Result Code	231
Sensor State Changed	0x0002	Ch	New State	232
Reference Found	0x0003	Ch	N/A	232
Following Error Limit	0x0004	Ch	N/A	233
Holding Aborted	0x0005	Ch	Result Code	231
Sensor Module State Changed	0x4000	Mod	New State	233
Over Temperature	0x4001	Mod	Temperature	233
High Voltage Overload	0x4002	Mod	N/A	234
Adjustment Finished	0x4010	Mod	Result Code	234
Adjustment State Changed	0x4011	Mod	New State	235
Adjustment Update	0x4012	Mod	Result Code	235
Stream Finished	0x8000	Dev	Stream Handle, Index, Result Code	235
Stream Ready	0x8001	Dev	Stream Handle	236
Stream Triggered	0x8002	Dev	Stream Handle	236
Command Group Triggered	0x8010	Dev	Transmit Handle, Res. Code	237
Hand Control Module State Changed	0x8020	Dev	New State	237
Emergency Stop Triggered	0x8030	Dev	N/A	238
External Input Triggered	0x8040	Dev	Input Index	238
Request Ready	0xf000	Any	Request ID, Request Type, Data Type, Array Size, Property Key	238

Continued on next page

Table 5.1 – *Continued from previous page*

Event	Code	Index	Parameter	Page
Connection Lost	0xf001	N/A	N/A	239

5.2 Detailed Event Description

5.2.1 None

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_NONE	0x0000	N/A	N/A			

Description:

This event type is a place holder indicating that no event occurred. The index and parameter fields are undefined.

5.2.2 Movement Finished

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_MOVEMENT_FINISHED	0x0001	Ch	Result Code			

Description:

This event is generated when a channel has finished a movement command (either successful or unsuccessful). See also section 2.6.7 "Movement Feedback".

Parameter:

The event parameter holds the result code. If the movement command finished successfully then the result is SA_CTL_ERROR_NONE. Otherwise the result code indicates what caused the failure. See table A.1 for a list of result codes.

5.2.3 Holding Aborted

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_HOLDING_ABORTED	0x0005	Ch	Result Code			

Description:

This event is generated when a channel detects an endstop (or a configured following error limit is exceeded) while in holding state.

Parameter:

The event parameter holds the result code: `SA_CTL_ERROR_END_STOP_REACHED` in case the holding was aborted due to an endstop or `SA_CTL_ERROR_FOLLOWING_ERR_LIMIT` in case the holding was aborted due to exceeding a following error limit.

5.2.4 Sensor State Changed

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
`SA_CTL_EVENT_SENSOR_STATE_CHANGED`	`0x0002`	Ch	New State			

Description:

A sensor was attached to or detached from a sensor module.

Parameter:

The parameter value will be one of:
`SA_CTL_EVENT_PARAM_ATTACHED` (`0x00000001`),
`SA_CTL_EVENT_PARAM_DETACHED` (`0x00000000`)

5.2.5 Reference Found

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
`SA_CTL_EVENT_REFERENCE_FOUND`	`0x0003`	Ch	N/A			

Description:

This event is generated during a reference movement. It is generated at the moment the physical position has been determined. Depending on the configuration of the referencing the movement might be continued and stopped at a later time. See section 2.6.2 "Referencing" for more information.

5.2.6 Following Error Limit

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_FOLLOWING_ERR_LIMIT	0x0004	Ch	N/A			

Description:

This event is generated if the configured following error limit is exceeded during a closed-loop movement. See section 2.11 "Following Error Detection" for more information.

5.2.7 Sensor Module State Changed

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_SM_STATE_CHANGED	0x4000	Mod	New State			

Description:

A sensor module was attached to or detached from a driver module.

Parameter:

The parameter value will be one of:
SA_CTL_EVENT_PARAM_ATTACHED (0x00000001),
SA_CTL_EVENT_PARAM_DETACHED (0x00000000)

5.2.8 Over Temperature

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_OVER_TEMPERATURE	0x4001	Mod	Temperature			

Description:

The module detected an over-temperature condition of a driver amplifier. Note that the amplifier circuit is automatically disabled at the occurrence of an over-temperature condition. The device must be cooled down before being able to continue to use the device. The Module State property (SA_CTL_MOD_STATE_BIT_OVER_TEMPERATURE) may be polled to know when the over temperature condition has passed by.

Parameter:

The parameter holds the measured temperature in °C.

5.2.9 High Voltage Overload

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_HIGH_VOLTAGE_OVERLOAD	0x4002	Mod	N/A			

Description:

The module detected an overload condition of the high voltage power supply.

5.2.10 Adjustment Finished

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_ADJUSTMENT_FINISHED	0x4010	Mod	Result Code			

Description:

This event is generated when a module adjustment process has finished (either successful or unsuccessful).

Parameter:

The event parameter holds the result code. If the adjustment finished successfully then the result is SA_CTL_ERROR_NONE. Otherwise the result code indicates what caused the failure. See table A.1 for a list of result codes.

5.2.11 Adjustment State Changed

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_ADJUSTMENT_STATE_CHANGED	0x4011	Mod	New State			

Description:

This event is generated when a module adjustment state changes.

Parameter:

The event parameter holds the new state of the adjustment process.

5.2.12 Adjustment Update

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_ADJUSTMENT_UPDATE	0x4012	Mod	Result Code			

Description:

This event is generated when a module adjustment update occurs.

Parameter:

The event parameter holds the result code. If the adjustment update finished successfully then the result is SA_CTL_ERROR_NONE. Otherwise the result code indicates what caused the failure. See table A.1 for a list of result codes.

5.2.13 Stream Finished

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_STREAM_FINISHED	0x8000	Dev	Handle	Index	Result Code	

Description:

This event indicates that a trajectory stream has come to an end. See section 2.15 "Trajectory Streaming" for more information.

Parameter:

The parameter holds information to further specify the event.

- **Stream Handle** The corresponding stream handle.
- **Index** The device/channel index that caused the given result code.
- **Result Code** The result of the trajectory streaming. If it finished successfully then the result is `SA_CTL_ERROR_NONE`. Otherwise the result code indicates what caused the failure. See table A.1 for a list of result codes.

5.2.14 Stream Ready

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
`SA_CTL_EVENT_STREAM_READY`	`0x8001`	Dev	Handle	Reserved		

Description:

This event indicates that the internal trajectory stream buffer contains enough data to start the stream. In case of direct streaming the stream will start automatically. Otherwise the device is ready to receive a start trigger for the stream. See section 2.15 "Trajectory Streaming" for more information.

Parameter:

The parameter holds the corresponding stream handle.

5.2.15 Stream Triggered

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
`SA_CTL_EVENT_STREAM_TRIGGERED`	`0x8002`	Dev	Handle	Reserved		

Description:

This event indicates that the controller has started to execute the trajectory stream. See section 2.15 "Trajectory Streaming" for more information.

Parameter:

The parameter holds the corresponding stream handle.

5.2.16 Command Group Triggered

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_CMD_GROUP_TRIGGERED	0x8010	Dev	Handle	Reserved	Result Code	

Description:

This event notifies that a command group has been executed (either directly or via a configured external trigger). See section 2.14 "Command Groups" for more information.

Parameter:

The parameter holds the corresponding transmit handle and result code.

5.2.17 Hand Control Module State Changed

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_SM_STATE_CHANGED	0x4000	Dev	New State			

Description:

A hand control module was attached to or detached from the device.

Parameter:

The parameter value will be one of:
SA_CTL_EVENT_PARAM_ATTACHED (0x00000001),
SA_CTL_EVENT_PARAM_DETACHED (0x00000000)

5.2.18 Emergency Stop Triggered

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_EMERGENCY_STOP_TRIGGERED	0x8030	Dev	N/A			

Description:

This event notifies that an emergency stop condition has been detected. See section 2.17.2 "Emergency Stop Mode" for more information.

5.2.19 External Input Triggered

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
SA_CTL_EVENT_EXT_INPUT_TRIGGERED	0x8040	Dev	Input Index			

Description:

This event notifies that an falling or rising edge was detected on the external trigger input. See section 2.17.5 "Event Trigger Mode" for more information.

Parameter:

The parameter holds the index of the input trigger (currently always 0).

5.2.20 Request Ready

Definition

C Definition	Code	Index	63-32			
SA_CTL_EVENT_REQUEST_READY	0xf000	Any	Property Key			
			31-24	23-16	15-8	7-0
			Size	Data Type	Rq. Type	Rq. ID

Description:

The request ready event is generated by the API when the result of an asynchronous request is received. The event is also generated in case of a request timeout or any other error. After the event has been received the result of the asynchronous operation can be retrieved using the `SA_CTL_ReadProperty_x`, `SA_CTL_WaitForWrite` functions. By waiting for this event, it is guaranteed that these functions won't block and return a result immediately. This event is not generated if the retrieve function for this request has already been called.

This event needs to be enabled using the Event Notification Options property.

Parameter:

The parameters store information needed to retrieve the result of the asynchronous request. The index parameter is same index as passed to the request function. Depending on the property key this is either a device, module or channel index.

- **Rq. ID** The request ID is identical to the one returned by the asynchronous request function and can be used to associate this event with open requests.
- **Rq. Type** The request type allows to differentiate between read and write requests. Possible values are `SA_CTL_EVENT_REQ_READY_TYPE_READ` (`0x00`) or `SA_CTL_EVENT_REQ_READY_TYPE_WRITE` (`0x01`)
- **Data Type** Indicates the type of the requested property. This information is needed to call the correct `SA_CTL_ReadProperty_x` function. If the property read failed, the data type is unknown and has a value of `SA_CTL_DTYPE_NONE` (`0xff`). In this case any of the `SA_CTL_ReadProperty_x` functions can be used to retrieve the error code.
- **Size** The array size stores the size of the received value. For integer properties this is the number of elements and for string properties the number of characters. Note that for strings the required buffer size is one byte larger because of the null terminator. This field is only set for successful property read requests.
- **Property Key** Key of the requested property.

Parameters can be extracted using the following macros:
`SA_CTL_EVENT_REQ_READY_ID()`,
`SA_CTL_EVENT_REQ_READY_TYPE()`,
`SA_CTL_EVENT_REQ_READY_DATA_TYPE()`,
`SA_CTL_EVENT_REQ_READY_ARRAY_SIZE()`,
`SA_CTL_EVENT_REQ_READY_PROPERTY_KEY()`

5.2.21 Connection Lost

Definition

C Definition	Code	Index	31-24	23-16	15-8	7-0
`SA_CTL_EVENT_CONNECTION_LOST`	`0xf001`	N/A	N/A			

Description:

The connection to the device has been lost. All functions requiring communication with the device will fail with `SA_CTL_ERROR_COMMUNICATION_FAILED`. After receiving this event the device should be closed using `SA_CTL_Close`.

6 *ASCII INTERFACE*

As an alternative to control the MCS2 using the SmarActCTL library, the device also supports control using an ASCII protocol. To simplify the entry and overall operation this protocol is (with some exceptions) strongly orientated towards the well established SCPI [1] standard.

NOTICE
The ASCII Interface is only available for devices with an ethernet port. For general information on how to configure the ethernet interface please refer to the *MCS2 User Manual* document.

6.1 Connection Setup

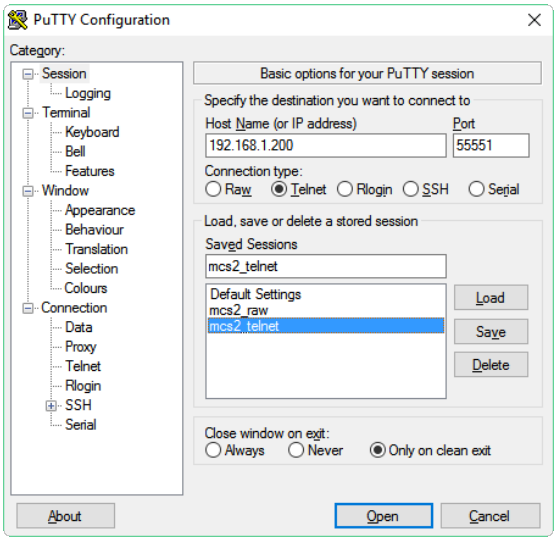
A connection to the device can be established via raw TCP/IP or by using a telnet client. The settings needed to access the ASCII Interface include:

- the current IP address (default is 192.168.1.200).
- the fixed port number 55551.

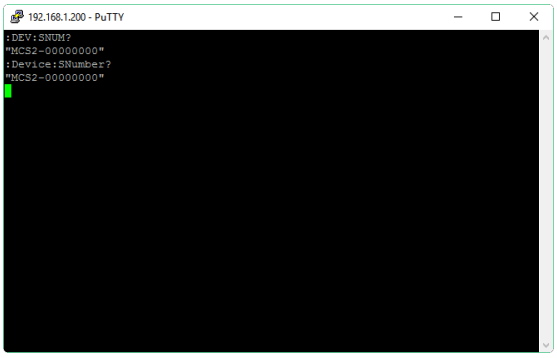
One way to connect and communicate with the device through the ASCII Interface is by using a telnet client. In the following steps we will use the multipurpose client PuTTY[2] to read the serial number of an attached MCS2 controller.

1. Download and start PuTTY (`www.putty.org`)
2. In the tree view to the left select the **session** category
3. Select **telnet** as connection type (see figure 6.1a)
4. Fill in the device's **IP** address and the correct **port** (55551)
5. Name and **save** the session options (optional)
6. A click on **open** will start the session (see figure 6.1b)
7. You are now ready to communicate with the device (e.g. to query the serial number).

[1] Standard Commands for Programmable Instruments (`www.ivifoundation.org/scpi`)
[2] Open source SSH and telnet client PuTTY (`www.putty.org`)



(a) PuTTY Configuration Window



(b) PuTTY Terminal Window

Figure 6.1: Communicating with the MCS2 using PuTTY

6.1.1 Note On Message Termination

When communicating with the device via raw TCP/IP make sure to use the correct message termination for commands sent to and answers received from the device. The message termination characters used by the MCS2 are <CR><LF> (carriage return + line feed).

6.2 SCPI Basics

Initially developed due to the need of a common interface language between computers and instruments, SCPI is nowadays a well established open standard to communicate with all kinds of devices. Due to it's easy to learn and mostly self-explanatory ASCII syntax it is usable with any computer language or application environment.

The following sections will give an overview on how to get started using SCPI with the MCS2. More information on the SCPI specification can be found on the IVI Foundation websites [3].

6.2.1 SCPI Conformance Information

Although being strongly orientated towards the SCPI standard (especially concerning the command syntax rules) we do not claim to be fully conform. Due to its rich set of functions and flexibility, the MCS2 does not fit in a predefined instrument class, but uses the well defined SCPI syntax and communication mechanisms for a convenient operation experience.

[3] www.ivifoundation.org/specifications/

6.2.2 Command Structure

SCPI differentiates between common and instrument commands. **Common commands** always start with an asterisk (*) and only consist of one keyword.

Common Command `*IDN?`

The behavior of these commands is mostly predefined by the standard and incorporates some general mechanisms like issuing a reset or reading global status bytes. Section 6.6.1 holds a table describing the common commands supported by the MCS2.

To access all the different properties and functions the MCS2 provides, **instrument commands** are used. These commands are device-dependent and follow a hierarchical tree system approach. Associated properties are therefore grouped into different subsystems (branches) creating a command tree like the one below.

```
[:PROPerty]          // "root"
    :DEVice          // "branch"
        :SNUMber     // "leaf"
        :STATe       // "leaf"
    :CHANnel#        // "branch"
        :VELocity    // "leaf"
```

As an example we now want to read the device's serial number. The assembling of a command always starts at the root of the tree. To obtain the value of a particular leaf the full path to it must be specified. This is achieved by traversing the command tree from root (`:PROPerty`) to leaf (`:SNUMber`) and concatenate the different keywords on the way from left to right. As result we get the full command string:

Instrument Command `:PROPerty:DEVice:SNUMber?`

Each command has both a **long and a short form**. Only the exact long or the exact short form will be accepted with lower- and uppercase letters being ignored (case-insensitive).

The following commands would all be accepted by the MCS2.

Long Form (mixed case)	`:PROPerty:DEVice:SNUMber?`
Long Form (all lower-case)	`:property:device:snumber?`
Short Form (all upper-case)	`:PROP:DEV:SNUM?`
Short Form (all lower-case)	`:prop:dev:snum?`

...

NOTICE

To keep track of long and short command forms, all of the following examples will use upper case letters for short commands and lower case letters for the remaining part of the corresponding long form.

A setup containing an MCS2 normally holds a variable number of channels and/or modules. To **address a particular module or channel**, the corresponding index has to be added when assembling the command. In general, if a command tree keyword contains a hash symbol (#) , that

symbol must be replaced by the desired module or channel index. Thus a `:CHANnel#` keyword becomes `:CHANnel2` when addressing the channel with index 2.

Many commands take an additional **command parameter** (e.g. to set a channel's velocity). Command and parameter must be separated by at least one space character. Command parameters can be of type numeric (int32/64) or type string and must be given according to the base unit (e.g. pm or n°).

The following command needs the channel's move velocity as a parameter given in $\frac{pm}{s}$.

Set velocity for channel 0 to 1 $\frac{mm}{s}$	`:PROPerty:CHANnel0:VELocity 1000000000`

For properties that are (also) readable, the **query form** of a command is generated by appending a question mark (?) to the command. However, not all commands have a query form, and some commands exist only in query form, see subsection 6.2.4 (Queries).

Query velocity for Channel 0	`:PROPerty:CHANnel0:VELocity?`
Response (in $\frac{pm}{s}$)	`1000000000`

6.2.3 Traversing the Command Tree

As stated in the previous section 6.2.2 (Command Structure) commands are created by concatenating keywords along the command tree. This section is intended to explain some more rules and possibilities on how to create proper commands.

- When assembling commands, **colons** (:) are used to separate the different keywords.
- **Square brackets** ([]) enclose a keyword that is optional (default) and may be omitted. Thus a command tree, starting with the root [:PROPerty] may lead to the following commands:
  - `:PROPerty:DEVice:SNUMber?`
  - `:DEVice:SNUMber?`
- Multiple commands may be sent in one message to the device (**compound command**).

The first command must always be referenced to the root node (e.g. :CHANnel0). Subsequent commands however, are referenced to the same tree level as the previous command in a message. These commands have to be separated by a semicolon (;) to the previous command.

Set channel 0 move mode	`:CHANnel0:MMODe 1`
Set channel 0 velocity	`:CHANnel0:VELocity 10000`
Set channel 0 acceleration	`:CHANnel0:ACCeleration 0`
Set all in one message	`:CHANnel0:MMODe 1;VELocity 10000;ACCeleration 0`
Set channel 0 positioner type	`:CHANnel0:PTYPe 300`

Note that sending a compound command message to the device may complicate error handling if one of the containing commands fails. It is therefore recommended to send each command as a single message to ensure a deterministic and stable program sequence.

6.2.4 Queries

To read the value of a specific device, module or channel property a query command has to be sent to the MCS2. Queries are generated by traversing the command tree and appending the final command with a question mark (?). When the device receives a valid query form of a command, a response is generated containing the current setting or value associated with the property.

Further note that

- query responses do not include the command header but only the requested value.
- for numeric properties, the result is returned as an int32/64 type (see Property Summary).
- for string properties, the result is returned as string.
- responses to compound query messages are separated by a semicolon (;).

Single query	`:CHANnel0:PTYPe?`
Response	`300`
Single query	`:CHANnel0:MMODe?`
Response	`2`
Compound Query	`:CHANnel0:PTYPe?;MMODe?`
Response	`300; 2`

To check whether a property is readable, writable or both, see section 6.6.3 (Property Command Tree).

6.3 Basic Programming Examples

This section shows a few examples how communication with the device might look using the short command forms and omitting the optional (default) `:PROPerty` command tree keyword. For more info on long and short command forms, see 6.2.2 (Command Structure). Note that commands are only executed after the device receives the <NL> character, see 6.1.1 (Note On Message Termination).

6.3.1 Get Property

```
// get number of bus modules from device
>> :DEV:NOMO?
// response
<< 1
```

6.3.2 Set Property

```
// set move mode to open-loop step mode (4) for channel 0
>> :CHAN0:MMOD 4
```

6.3.3 Calibrate

```
// set calibration mode for channel 0 (start direction: forward)
>> :CHAN0:CAL:OPT 0
// start calibration sequence
>> :CAL0
```

6.3.4 Reference

```
// set find reference mode for channel 0 (default is 0)
>> :CHAN0:REF:OPT 0
// start referencing sequence
>> :REF0
```

6.3.5 Move

```
// set move mode to closed-loop relative (1) for channel 0
>> :CHAN0:MMOD 1
// set move velocity [in pm/s]
>> :CHAN0:VEL 500000000
// disable acceleration control
>> :CHAN0:ACC 0
// start actual movement, value is interpreted as
// relative position (in pm)
>> :MOVE0 500000000
```

6.3.6 Stop

```
// send stop command to channel 0
>> :STOP0
```

6.3.7 Movement State

```
// get current state for channel 0
>> :CHAN0:STAT?
// response holds the state bitmask as int32 value
<< 37
// decoding the value leads us to the following active state bits
// - channel 0 is actively moving   (bit 0 is set)
// - channel 0 is calibrating       (bit 2 is set)
// - channel 0 has a sensor present (bit 5 is set)
```

6.3.8 Error Handling

To access information on errors due to either incorrect assembling of command messages or general handling with the device, the ASCII Interface holds a user accessible error queue.

This queue is implemented as FIFO[4] and can be accessed by the `:SYSTem:ERRor` subsystem. Errors that occur during run-time can therefore be detected by executing the following queries.

`:SYSTem:ERRor:COUNt?`	Returns the number of errors the queue contains
`:SYSTem:ERRor[:NEXT]?`	Returns the NEXT error and removes it from the queue (will return `0, "No Error"` if empty)

Error codes returned are divided in

- a No Error Code which is equal to zero.
- SCPI error codes which are less than zero, see 6.4.
- and SmarActControl error codes which are greater than zero, see A.1.

A program sequence with error checking might look like the following:

```
// try to get current state for channel 0
>> :CHAN0:STAT?!
// due to an invalid character in this command (!), there is no response
// by checking the error count
>> :SYST:ERR:COUN?
// we see that there is one error inside the error queue
<< 1
// to get more information we retrieve this error
>> :SYST:ERR:NEXT?
// and get the following response
<< -101,"Invalid character"
```

[4] **F**irst error **I**n will be the **F**irst error **O**ut

NOTICE

Note that when working with the error queue, it might already hold errors generated by previous commands. An incorrect command can even result in multiple errors being added to the queue. It is therefore good practice to query all possible errors before sending the next command.

6.4 Using Command Groups

Command groups offer the possibility to define an atomic group of commands that is executed synchronously. In addition, a command group may not only be triggered via software, but alternatively via an external trigger. For more general information on Command Groups please refer to section 2.14.

This section describes how to take advantage of Command Groups when using the ASCII interface.

6.4.1 Command Set

The following commands and queries are used to control a Command Group.

:CGRoup:OPEN <triggerMode>	Opens a Command Group using the given trigger mode.
:CGRoup:CLOSe	Closes a previously opened Command Group.
:CGRoup:ABORt	Aborts a previously opened Command Group.
:CGRoup:FINished?	Indicates whether the Command Group is finished.
:CGRoup:VALues?	Requests the values that were queried inside a Command Group.

Note that, when using the ASCII interface, the number of concurrently active Command Groups is limited to one. Figure 6.2 show the general process for either writing or reading multiple properties using a Command Group.

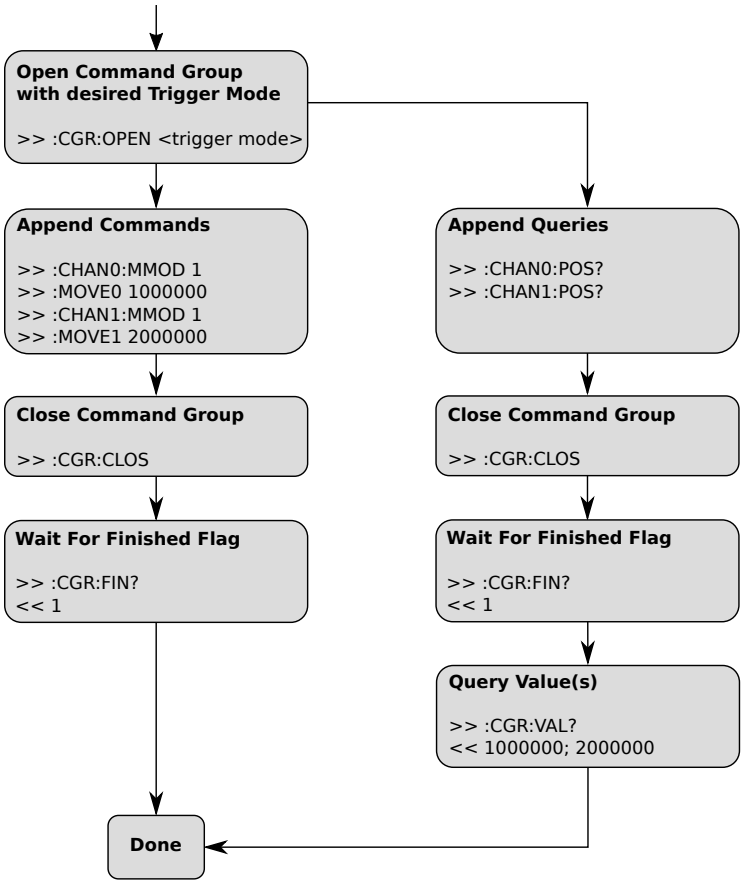


Figure 6.2: Command Group procedure(s)

The CGR:OPEN command is used to activate a Command Group using the given trigger mode. All of the following commands and queries will be appended to this Command Group. Note that properties missing the *Groupable* flag will lead to an error when put into a Command Group. Sending the CGR:CLOS command either starts the Command Group's execution immediately (trigger mode direct) or defers the execution until an external event occurs (trigger mode external).

The CGR:FIN query is used to check if execution of all grouped commands has been started or if the requested values are available.

For finished Command Groups that contained at least one query, the CGR:VAL query is used to read the resulting values from the device.

6.4.2 Examples

This section contains some examples to further demonstrate the different use cases of Command Groups.

Synchronized movement using direct trigger

The following sequence uses a Command Group to synchronize the Closed-Loop movement of two channels. By using the Direct Trigger mode, the commands execution starts right after closing the Command Group.

```
// open command group in direct trigger mode (0)
// (every following command is not executed but put into the group)
>> :CGR:OPEN 0
// set move modes of channel 0 and 1 to closed-loop relative (1)
>> :CHAN0:MMOD 1
>> :CHAN1:MMOD 1
// move channel 0 to +1mm
>> :MOVE0 1000000000
// move channel 1 to +0.5mm
>> :MOVE1 500000000
// close command group
// (execution of grouped commands starts now)
>> :CGR:CLOS
// the command group's finished value signalizes
// that the command group has been processed
>> :CGR:FIN?
<< 1
```

Synchronized position query using direct trigger

The following sequence uses a Command Group to synchronize the position sampling of two channels. By using the Direct Trigger mode, the queries' execution starts right after closing the Command Group.

```
// open command group in direct trigger mode (0)
// (every following query is not executed but put into the group)
>> :CGR:OPEN 0
// query positions of channel 0 and 1
>> :CHAN0:POS?
>> :CHAN1:POS?
// close command group
// (execution of grouped commands starts now)
>> :CGR:CLOS
// the command group's finished value signalizes
// that the command group has been processed
>> :CGR:FIN?
<< 1
// we can now query the resulting value(s)
>> :CGR:VAL?
<< 1000000000; 500000000
```

Synchronized movement using external trigger

The following sequence uses a Command Group to synchronize the Closed-Loop movement of two channels. By using the External Trigger mode, the commands execution is deferred until the external event occurs. Note that the Input Trigger has to be configured accordingly in advance. See section 2.17 for more information.

```
// open command group in external trigger mode (1)
// (every following command is not executed but put into the group)
>> :CGR:OPEN 1
// set move modes of channel 0 and 1 to closed-loop relative (1)
>> :CHAN0:MMOD 1
>> :CHAN1:MMOD 1
// move channel 0 to +1mm
>> :MOVE0 1000000000
// move channel 1 to +0.5mm
>> :MOVE1 500000000
// close command group
// (execution of grouped commands is deferred)
>> :CGR:CLOS
// the command group's finished value signalizes
// that the command group has NOT been processed yet
>> :CGR:FIN?
<< 0
// ...
// process some other commands/queries
// ...
-> external event occurs
// the command group's finished value signalizes
// that the command group has now been processed
>> :CGR:FIN?
<< 1
```

6.5 Streaming Trajectories

Trajectory streaming allows a multi DoF manipulator to follow specific trajectories using the MCS2 controller. All participating positioners are moved synchronously along the defined trajectory. For more general information on Streaming please refer to 2.15.

This section describes how to take advantage of Trajectory Streaming when using the ASCII interface.

6.5.1 Command Set

The following commands and queries are used to control a trajectory stream.

:STReam:OPEN <triggerMode>	Opens a stream using the given trigger mode.
:STReam:BFREe?	Returns the number of free buffer slots.
:STReam:FRAMe <frameData>	Transmits the desired frame.
:STReam:CLOSe	Closes a running stream.
:STReam:ABORt	Aborts a running stream.

Before starting a stream make sure to configure the properties below as desired:

Stream Base Rate Configures the stream base rate in Hz, see 197.

Stream Sync Rate Configures the external synchronization rate in Hz, see 198.

Stream Options Configures the stream behavior, see 199.

NOTICE

When using the ASCII interface, the maximum reachable streaming frequency is reduced, depending on the number of involved channels and the programming sequence.
To prevent buffer under-/overruns, make sure to always supply enough stream frames according to the remaining free buffer slots.

Figure 6.3 shows the general procedure for a complete streaming sequence.

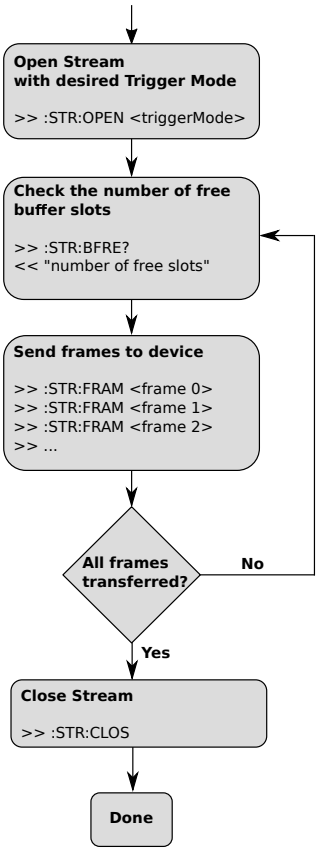


Figure 6.3: Streaming sequence

The `STR:OPEN` command is used to open a stream using the given trigger mode.

By reading the number of available buffer slots using the `STR:BFRE` query, the number of frames that can currently be transferred to the device can be calculated. The number of free buffer slots is given in positions, thus a stream containing two channels would take up two buffer slots. Using the `STR:FRAM` command, the device is now provided with the desired positions for each channel. A frame is assembled using a channel index following the corresponding absolute position, separated by comma. This mechanism is used until all frames have been sent to the device.

The `STR:CLOS` command is used to close the stream.

6.5.2 Example

The following example configures and sends a stream to the device containing positions for channel 0 and 1.

```
// configure the streaming base rate to 100Hz
>> :DEV:STR:BAS 100
// configure the streaming options to default (0)
>> :DEV:STR:OPT 0
```

```
// open stream in direct trigger mode (0)
>> :STR:OPEN 0
// check the current buffer level
>> :STR:BFRE?
<< 1024
// We have 1024 position buffer slots available.
// (This effectively results in 1024/numberOfChannels=512 frame slots)
// Now we transmit our frames containing positions for channel 0 and 1.
>> STR:FRAM 0,1000000,1,100000
>> STR:FRAM 0,2000000,1,150000
>> STR:FRAM 0,3000000,1,200000
>> ...
// Streaming starts as soon as enough data has been received by the
// device. Repeat this process until all desired frames have been
// sent to the device.
// If all frames have been transferred, close the stream.
>> :STR:CLOS
// The remaining frames are processed until the stream is completed.
```

6.6 Command Summary

Section 6.6.1 contains an overview of the supported set of SCPI common commands and their behavior in context of the MCS2. The following tables in section 6.2 and 6.3 show the command hierarchy as well as the necessary information to assemble all instrument commands available through the ASCII Interface.

6.6.1 Common Commands

In general, the ASCII Interface supports all mandatory common commands required by the SCPI standard. Nevertheless most of them are not needed for controlling the device. Table 6.1 shows an overview of the implemented common commands and their utilization.

Table 6.1 – Common Commands

Mnemonic	Name	Description
*CLS	Clear Status Command	This command clears all status data structures.
*ESE	Standard Event Status Enable Command	This command has no effect.
*ESE?	Standard Event Status Enable Query	This command has no effect.
*ESR	Standard Event Status Register Query	This command has no effect.
*IDN?	Identification Query	This command returns information about the device such as manufacturer and serial number.

Continued on next page

Table 6.1 – *Continued from previous page*

Mnemonic	Name	Description
*OPC	Operation Complete Command	This command has no effect.
*OPC?	Operation Complete Query	This command has no effect (will always return 1).
*RST	Reset Command	Resets the device (reconnect necessary!).
*SRE	Service Request Enable Command	This command has no effect.
*SRE?	Service Request Enable Query	This command has no effect.
*STB?	Read Status Byte Query	Returns the status byte.
*TST?	Self-Test Query	This command has no effect (will always return 0).
*WAI	Wait-to-Continue Command	This command has no effect.

6.6.2 Movement Commands

Table 6.2 shows the commands that generate or stop movement. For detailed information on a movement command please follow the corresponding page to the Function Reference chapter.

Table 6.2 – Movement Summary

SCPI Command Tree	Type	Idx	Access	Page
`:MOVE#`	I64	Ch	W	122
`:STOP#`	-	Ch	W	124
`:CALibrate#`	-	Ch	W	118
`:REFerence#`	-	Ch	W	120

6.6.3 Property Command Tree

Table 6.3 shows the command hierarchy to access all the properties available for a proper device configuration. For detailed information on a property please follow the corresponding page to the Property Reference chapter.

Table 6.3 – Property Summary

SCPI Command Tree	Type	Idx	Access	Property	Page
`[:PROPerty]`					
`:DEVice`					
`:NOCHannels`	I32	Dev	R	Number of Channels	137

Continued on next page

Table 6.3 – *Continued from previous page*

SCPI Command Tree	Type	Idx	Access	Property	Page
:NOBModules	I32	Dev	R	Number of Bus Modules	137
:STATe	I32	Dev	R	Device State	138
:SNUMber	String	Dev	R	Device Serial Number	139
:NAME	String	Dev	R	Device Name	140
:ESTop					
:MODE	I32	Dev	RW	Emergency Stop Mode	141
:NETWork					
:DISCover					
:MODE	I32	Dev	RW	Network Discover Mode	142
:STReaming					
:BASerate	I32	Dev	RW	Stream Base Rate	197
:SYNCrate	I32	Dev	RW	Stream External Sync Rate	198
:OPTions	I32	Dev	RW	Stream Options	199
:HMODule					
:LOPTions					
[:CURRent]	I32	Dev	RW	Hand Control Module Lock Options	224
:DEFault	I32	Dev	RW	Hand Control Module Default Lock Options	226
:TRIGger					
:INPut					
:MODE	I32	Dev	RW	Device Input Trigger Mode	215
:CONDition	I32	Dev	RW	Device Input Trigger Condition	216
:MODule#					
:PSUPply					
[:ENABled]	I32	Mod	RW	Power Supply Enabled	143
:STATe	I32	Mod	R	Module State	144
:NOMChannels	I32	Mod	R	Number of Bus Module Channels	145
:TEMPerature	I32	Mod	R	Bus Module Temperature	202
:IOModule					
:OPTions	I32	Mod	RW	I/O Module Options	211
:VOLTage	I32	Mod	RW	I/O Module Voltage	213
:AINPut					
:RANGe	I32	Mod	RW	I/O Module Analog Input Range	213
:AUXiliary					

Continued on next page

Table 6.3 – *Continued from previous page*

SCPI Command Tree	Type	Idx	Access	Property	Page
:DINPut					
[:VALue]	I32	Mod	R	Aux Digital Input Value	208
:DOUTput					
[:VALue]	I32	Mod	RW	Aux Digital Output Value / Set / Clear	209
:SET	I32	Mod	RW	Aux Digital Output Value / Set / Clear	209
:CLEar	I32	Mod	RW	Aux Digital Output Value / Set / Clear	209
:AOUTput					
[:VALue]#	I32	Mod	RW	Aux Analog Output Value0 / Value1	210
:CHANnel#					
:AMPLifier					
[:ENABled]	I32	Ch	RW	Amplifier Enabled	145
:PCONtrol					
:OPTions	I32	Ch	RW	Positioner Control Options	146
:ACTuator					
:MODE	I32	Ch	RW	Actuator Mode	147
:CLINput					
[:SELect]	I32	Ch	RW	Control Loop Input	149
:SENSor					
:SELect	I32	Ch	RW	Sensor Input Select	150
[:VALue]	I64	Ch	R	Control Loop Input Sensor Value	170
:AUXiliary					
[:VALue]	I64	Ch	R	Control Loop Input Aux Value	171
:PTYPe					
[:CODE]	I32	Ch	RW	Positioner Type	151
:NAME	String	Ch	R	Positioner Type Name	152
:MMODe	I32	Ch	RW	Move Mode	152
:STATe	I32	Ch	R	Channel State	154
:POSition					
[:CURRent]	I64	Ch	RW	Position	155
:TARGet	I64	Ch	R	Target Position	156
:SCAN	I64	Ch	R	Scan Position	156
:MSHift	I32	Ch	RW	Position Mean Shift	168
:SCAN					

Continued on next page

Table 6.3 – *Continued from previous page*

SCPI Command Tree	Type	Idx	Access	Property	Page
:VELocity	I64	Ch	RW	Scan Velocity	157
:HOLDtime	I32	Ch	RW	Hold Time	158
:VELocity	I64	Ch	RW	Move Velocity	159
:ACCeleration	I64	Ch	RW	Move Acceleration	160
:MCLFrequency					
[:CURRent]	I32	Ch	RW	Max Closed Loop Frequency	161
:DEFault	I32	Ch	RW	Default Max Closed Loop Frequency	162
:STEP					
:FREQuency	I32	Ch	RW	Step Frequency	163
:AMPLitude	I32	Ch	RW	Step Amplitude	163
:FELimit	I64	Ch	RW	Following Error Limit	164
:BSTop					
:OPtions	I32	Ch	RW	Broadcast Stop Options	165
:SENSor					
:PMODe	I32	Ch	RW	Sensor Power Mode	166
:PSDelay	I32	Ch	RW	Sensor Power Save Delay	167
:SDIRection	I32	Ch	RW	Safe Direction	169
:LSCale					
:OFFset	I64	Ch	RW	Logical Scale Offset	173
:INVersion	I32	Ch	RW	Logical Scale Inversion	174
:RLIMit					
:MIN	I64	Ch	RW	Range Limit Min	175
:MAX	I64	Ch	RW	Range Limit Max	175
:CALibration					
:OPTions	I32	Ch	RW	Calibration Options	176
:SCORrection					
:OPTions	I32	Ch	RW	Signal Correction Options	177
:REFerencing					
:OPTions	I32	Ch	RW	Referencing Options	179
:DTRMark	I32	Ch	RW	Distance To Reference Mark	180
:DCINverted	I32	Ch	RW	Distance Code Inverted	180
:ERRor	I32	Ch	R	Channel Error	200

Continued on next page

Table 6.3 – *Continued from previous page*

SCPI Command Tree	Type	Idx	Access	Property	Page
:TEMPerature	I32	Ch	R	Channel Temperature	201
:TTZVoltage					
:THReshold	I32	Ch	RW	Target To Zero Voltage Hold Threshold	172
:AUXiliary					
:PTYPe	I32	Ch	RW	Aux Positioner Type	203
:PTName	String	Ch	R	Aux Positioner Type Name	204
:ISELect	I32	Ch	RW	Aux Input Select	204
:IOModule					
:INPut					
:INDex	I32	Ch	RW	Aux I/O Module Input Index	205
[:VALue]#	I32	Ch	R	Aux I/O Module Input0 / Input1 Value	208
:DINVersion	I32	Ch	RW	Aux Direction Inversion	207
:TRIGger					
:OUTPut					
:MODE	I32	Ch	RW	Channel Output Trigger Mode	217
:POLarity	I32	Ch	RW	Channel Output Trigger Polarity	218
:PWidth	I32	Ch	RW	Channel Output Trigger Pulse Width	219
:PCOMpare					
:THReshold					
[:STARt]	I32	Ch	RW	Channel Position Compare Start Threshold	220
:INCRement	I32	Ch	RW	Channel Position Compare Increment	221
:DIRection	I32	Ch	RW	Channel Position Compare Direction	221
:LMIN	I64	Ch	RW	Channel Position Compare Limit Min	222
:LMAX	I64	Ch	RW	Channel Position Compare Limit Max	223
:TUNing					
:MTYPe	I32	Ch	R(W)	Positioner Movement Type	181
:CUSTom	I32	Ch	R(W)	Positioner Is Custom Type	182
:BASE					
:UNIT	I32	Ch	R(W)	Positioner Base Unit	183
:RESolution	I32	Ch	R(W)	Positioner Base Resolution	184
:HTYPe	I32	Ch	R(W)	Positioner Sensor Head Type	185
:RTYPe	I32	Ch	R(W)	Positioner Reference Type	186

Continued on next page

Table 6.3 – *Continued from previous page*

SCPI Command Tree	Type	Idx	Access	Property	Page
:GAIN					
:P	I32	Ch	R(W)	Positioner P Gain	187
:I	I32	Ch	R(W)	Positioner I Gain	188
:D	I32	Ch	R(W)	Positioner D Gain	189
:SHIFt	I32	Ch	R(W)	Positioner PID Shift	190
:AWINdup	I32	Ch	R(W)	Positioner Anti Windup	191
:SAVE	I32	Ch	W	Save Positioner Type	196
:WPRotection	I32	Ch	RW	Positioner Write Protection	196
:ESDetection					
:DISTance	I32	Ch	R(W)	Positioner ESD Distance Threshold	192
:COUNter	I32	Ch	R(W)	Positioner ESD Counter Threshold	193
:THReshold					
:TREached	I32	Ch	R(W)	Positioner Target Reached Threshold	194
:THOLd	I32	Ch	R(W)	Positioner Target Hold Threshold	195

6.7 SCPI Error Codes

Table 6.4 – SCPI Error Codes

Code	Definition / Description
0	SCPI_ERROR_NO_ERROR No error occurred. Corresponds to an acknowledge.
-101	SCPI_ERROR_INVALID_CHARACTER The command message contained an invalid character.
-103	SCPI_ERROR_INVALID_SEPARATOR The command message contained an invalid separator.
-104	SCPI_ERROR_DATA_TYPE_ERROR The command message contained an illegal data type.
-108	SCPI_ERROR_PARAMETER_NOT_ALLOWED The command message contained illegal parameter.
-109	SCPI_ERROR_MISSING_PARAMETER The command message is missing a parameter.
-113	SCPI_ERROR_UNDEFINED_HEADER The command message does not exist for this device.

Continued on next page

Table 6.4 – *Continued from previous page*

Code	Definition / Description
-151	SCPI_ERROR_INVALID_STRING_DATA The given string data is invalid.
-350	SCPI_ERROR_QUEUE_OVERFLOW An internal error queue overflow occurred.
-363	SCPI_ERROR_INPUT_BUFFER_OVERRUN An input buffer overrun occurred.

A *CODE DEFINITION REFERENCE*

A.1 Error Codes

Table A.1 – Error Codes

Code	C-Definition / Description
0x0000	SA_CTL_ERROR_NONE No error occurred. Corresponds to an acknowledge.
0x0001	SA_CTL_ERROR_UNKNOWN_COMMAND An unknown command opcode was received and the packet was dropped.
0x0002	SA_CTL_ERROR_INVALID_PACKET_SIZE Indicates that the size field of a packet does not match the packet structure.
0x0004	SA_CTL_ERROR_TIMEOUT A timeout occurred while receiving a complete packet.
0x0005	SA_CTL_ERROR_INVALID_PROTOCOL A packet was received that does not comply to a supported protocol.
0x000c	SA_CTL_ERROR_BUFFER_UNDERFLOW The targeted buffer is empty.
0x000d	SA_CTL_ERROR_BUFFER_OVERFLOW The targeted buffer is filled and has no more space for further data.
0x000e	SA_CTL_ERROR_INVALID_FRAME_SIZE The frame size of the packet is invalid.
0x0010	SA_CTL_ERROR_INVALID_PACKET A packet with an inconsistent structure was received.
0x0012	SA_CTL_ERROR_INVALID_KEY The given property key could not be resolved.
0x0013	SA_CTL_ERROR_INVALID_PARAMETER The passed parameter is not in the valid range.
0x0016	SA_CTL_ERROR_INVALID_DATA_TYPE Indicates that the data type of a parameter is invalid.
0x0017	SA_CTL_ERROR_INVALID_DATA The command could not be processed due to invalid data. (E.g. a calibration routine finished but could not generate valid data.)

Continued on next page

Table A.1 – *Continued from previous page*

Code	C-Definition / Description
0x0018	SA_CTL_ERROR_HANDLE_LIMIT_REACHED The command could not be processed because all available handles are currently in use.
0x0019	SA_CTL_ERROR_ABORTED The command has been aborted.
0x0020	SA_CTL_ERROR_INVALID_DEVICE_INDEX An invalid device index has been passed.
0x0021	SA_CTL_ERROR_INVALID_MODULE_INDEX An invalid module index has been passed.
0x0022	SA_CTL_ERROR_INVALID_CHANNEL_INDEX An invalid channel index has been passed.
0x0023	SA_CTL_ERROR_PERMISSION_DENIED The request cannot be processed due to an access violation.
0x0024	SA_CTL_ERROR_COMMAND_NOT_GROUPABLE The given command cannot be part of a command group.
0x0025	SA_CTL_ERROR_MOVEMENT_LOCKED The given command cannot be processed due to movements being locked.
0x0026	SA_CTL_ERROR_SYNC_FAILED A synchronization requirement could not be met. (E.g. the trajectory streaming was aborted due to a stream overload.)
0x0027	SA_CTL_ERROR_INVALID_ARRAY_SIZE The number of array elements is invalid for a given write array property command.
0x0028	SA_CTL_ERROR_OVERRANGE An over-range condition occurred.
0x0029	SA_CTL_ERROR_INVALID_CONFIGURATION The operation could not be started due to an invalid configuration of the component. (E.g. some other properties are not configured properly for the configured operation mode.)
0x0100	SA_CTL_ERROR_NO_HM_PRESENT The command could not be processed because no Hand-Control-Module is present.
0x0101	SA_CTL_ERROR_NO_IOM_PRESENT The command could not be processed because no I/O-Module is present.
0x0102	SA_CTL_ERROR_NO_SM_PRESENT The command could not be processed because no Sensor-Module is present.
0x0103	SA_CTL_ERROR_NO_SENSOR_PRESENT The command could not be processed because no sensor is present.
0x0104	SA_CTL_ERROR_SENSOR_DISABLED The command could not be processed because the sensor is disabled.

Continued on next page

Table A.1 – *Continued from previous page*

Code	C-Definition / Description
0x0105	SA_CTL_ERROR_POWER_SUPPLY_DISABLED The command could not be processed because the power supply is disabled.
0x0106	SA_CTL_ERROR_AMPLIFIER_DISABLED The command could not be processed because the amplifier is disabled.
0x0107	SA_CTL_ERROR_INVALID_SENSOR_MODE The command could not be processed with the current sensor mode setting. (E.g. the power save mode is not allowed while trajectory streaming.)
0x0108	SA_CTL_ERROR_INVALID_ACTUATOR_MODE The command could not be processed with the current actuator mode setting.
0x0109	SA_CTL_ERROR_INVALID_INPUT_TRIG_MODE The command could not be processed with the current input trigger mode setting.
0x010a	SA_CTL_ERROR_INVALID_CONTROL_OPTIONS The command could not be processed with the current control options setting.
0x010b	SA_CTL_ERROR_INVALID_REFERENCE_TYPE The command could not be processed with the current reference type of the positioner.
0x010c	SA_CTL_ERROR_INVALID_ADJUSTMENT_STATE The command could not be processed with the current adjustment state.
0x010e	SA_CTL_ERROR_NO_FULL_ACCESS The command could not be processed because the MCS2 has not full access connection to a connected Picoscale sensor.
0x010f	SA_CTL_ERROR_ADJUSTMENT_FAILED An adjustment sequence failed.
0x0110	SA_CTL_ERROR_MOVEMENT_OVERRIDDEN A software commands a movement which is then interrupted by the Hand Control Module before it finished or vice versa.
0x0111	SA_CTL_ERROR_NOT_CALIBRATED The command could not be processed because the channel is not calibrated.
0x0112	SA_CTL_ERROR_NOT_REFERENCED The command could not be processed because the channel is not referenced.
0x0113	SA_CTL_ERROR_NOT_ADJUSTED The command could not be processed because the channel is not adjusted.
0x0114	SA_CTL_ERROR_SENSOR_TYPE_NOT_SUPPORTED The command could not be processed because the sensor type of the configured positioner is not supported from the hardware (e.g. from the sensor module).
0x0115	SA_CTL_ERROR_CONTROL_LOOP_INPUT_DISABLED The command could not be processed because the control-loop input is disabled. (See Control Loop Input property.)

Continued on next page

Table A.1 – *Continued from previous page*

Code	C-Definition / Description
0x0116	SA_CTL_ERROR_INVALID_CONTROL_LOOP_INPUT The command could not be processed because the control-loop input is invalid for the command. (E.g. the calibration and referencing movements cannot be started when the control-loop input is configured to 'aux in'.)
0x0117	SA_CTL_ERROR_UNEXPECTED_SENSOR_DATA The calibration routine could not be processed due to unexpected data from the position sensor.
0x0150	SA_CTL_ERROR_BUSY_MOVING The command could not be processed because the channel is currently busy performing a movement command. (E.g. disabling the velocity control while moving is not permitted.)
0x0151	SA_CTL_ERROR_BUSY_CALIBRATING The command could not be processed because the channel is currently busy performing a calibration sequence.
0x0152	SA_CTL_ERROR_BUSY_REFERENCING The command could not be processed because the channel is currently busy performing a referencing sequence.
0x0153	SA_CTL_ERROR_BUSY_ADJUSTING The command could not be processed because the channel is currently busy performing an adjustment sequence.
0x0200	SA_CTL_ERROR_END_STOP_REACHED An endstop was detected.
0x0201	SA_CTL_ERROR_FOLLOWING_ERR_LIMIT The following error exceeded the configured limit.
0x0202	SA_CTL_ERROR_RANGE_LIMIT_REACHED A configured position limit was hit.
0x0300	SA_CTL_ERROR_INVALID_STREAM_HANDLE The given stream handle is invalid.
0x0301	SA_CTL_ERROR_INVALID_STREAM_CONFIGURATION The configured streaming parameters are not supported by all modules.
0x0302	SA_CTL_ERROR_INSUFFICIENT_FRAMES This error is generated if the trajectory streaming was started without providing the minimum amount of frames. (A trajectory stream must consist of at least two frames.)
0x0303	SA_CTL_ERROR_BUSY_STREAMING The command could not be processed because the channel is currently participating in a trajectory stream.
0x0400	SA_CTL_ERROR_HM_INVALID_SLOT_INDEX An invalid slot index has been passed to the hand control module.

Continued on next page

Table A.1 – *Continued from previous page*

Code	C-Definition / Description
0x0401	SA_CTL_ERROR_HM_INVALID_CHANNEL_INDEX An invalid channel index has been passed to the hand control module.
0x0402	SA_CTL_ERROR_HM_INVALID_GROUP_INDEX An invalid group index has been passed to the hand control module.
0x0403	SA_CTL_ERROR_HM_INVALID_CH_GRP_INDEX An invalid channel group index has been passed to the hand control module.
0x0500	SA_CTL_ERROR_INTERNAL_COMMUNICATION An internal communication error occurred.
0x7ffd	SA_CTL_ERROR_FEATURE_NOT_SUPPORTED Indicates that a requested feature is not available on the connected device.
0x7ffe	SA_CTL_ERROR_FEATURE_NOT_IMPLEMENTED Indicates that a feature is not yet implemented. The device may have to be update to a newer version.
0xf000	SA_CTL_ERROR_DEVICE_LIMIT_REACHED The maximum number of devices has been opened.
0xf001	SA_CTL_ERROR_INVALID_LOCATOR An invalid locator string has been passed.
0xf002	SA_CTL_ERROR_INITIALIZATION_FAILED Initialization of the desired device failed.
0xf003	SA_CTL_ERROR_NOT_INITIALIZED The device has not been initialized yet.
0xf004	SA_CTL_ERROR_COMMUNICATION_FAILED Communication with the device failed.
0xf006	SA_CTL_ERROR_INVALID_QUERYBUFFER_SIZE The provided array size does not meet the required size.
0xf007	SA_CTL_ERROR_INVALID_DEVICE_HANDLE An invalid device handle has been passed.
0xf008	SA_CTL_ERROR_INVALID_TRANSMIT_HANDLE An invalid transmit handle has been passed.
0xf00f	SA_CTL_ERROR_UNEXPECTED_PACKET_RECEIVED An unexpected packet has been received.
0xf010	SA_CTL_ERROR_CANCELED The function call has been canceled.
0xf013	SA_CTL_ERROR_DRIVER_FAILED The device could not be found due to a driver failure.
0xf016	SA_CTL_ERROR_BUFFER_LIMIT_REACHED The limit of available buffers has been reached.

Continued on next page

Table A.1 – *Continued from previous page*

Code	C-Definition / Description
0xf017	SA_CTL_ERROR_INVALID_PROTOCOL_VERSION A protocol version mismatch has been detected.
0xf018	SA_CTL_ERROR_DEVICE_RESET_FAILED The device software reset failed.
0xf019	SA_CTL_ERROR_BUFFER_EMPTY Action is not allowed with empty buffers (e.g. empty command group buffer).
0xf01a	SA_CTL_ERROR_DEVICE_NOT_FOUND The device specified in the locator could not be found.
0xf01b	SA_CTL_ERROR_THREAD_LIMIT_REACHED The maximum number of simultaneous calls for this function was reached.

Sales partner / Contacts

Headquarters

SmarAct GmbH

Schuette-Lanz-Strasse 9
26135 Oldenburg
Germany

T: +49 441 – 800 87 90
Email: info-de@smaract.com
www.smaract.com

France

SmarAct GmbH

Schuette-Lanz-Strasse 9
26135 Oldenburg
Germany

T: +49 441 – 80 08 79 956
Email: info-fr@smaract.com
www.smaract.com

Israel

Trico Israel Ltd.

P.O.Box 6172
46150 Herzeliya
Israel

T: +972 9 – 950 60 74
Email: info-il@smaract.com
www.trico.co.il

Japan

Physix Technology Inc.

Ichikawa-Business-Plaza
4-2-5 Minami-yawata,
Ichikawa-shi
272-0023 Chiba
Japan
T/F: +81 47 – 370 86 00
Email: info-jp@smaract.com
www.physix-tech.com

South Korea

SEUM Tronics

801, 1, Gasan digital 1-ro
Geumcheon-gu
Seoul, 08594,
Korea

T: +82 2 868 – 10 02
Email: info-kr@smaract.com
www.seumtronics.com

USA

SmarAct Inc.

2140 Shattuck Ave. Suite 1103
Berkeley, CA 94704
United States of America

T: +1 415 – 766 9006
Email: info-us@smaract.com
www.smaract.com