MCS - Modular Control System
Programmer's Guide

SmarAct GmbH
Schuette-Lanz-Strasse 9
D-26135 Oldenburg

Tel.: +49 (0) 441 8008 79-0
Fax: +49 (0) 441 8008 79-21

eMail: info@smaract.de
www.smaract.de

Document Version 14-10-09-604
Library Version 2.0.11

Table of Contents

1 Introduction

This document describes the usage of the MCSControl function library which is used to control one or more MCS controller by software. It may be used for integration into existing software environments (e.g. LabVIEW) or to provide access to the system when writing your own software. The header file that comes along with the library summarizes the functions of the library and lists definitions like error and status codes.

While chapter 3 describes the library functions in detail, chapter 2 gives additional information on how to use various features of the MCS and which functions to use for these features.

Chapter 4 gives an overview of all available functions of the library along with a short description. This will give you a better orientation on which functions to use to achieve your goals.

Chapter 5 summarizes the most common definitions like status and error codes.

You may connect several MCS controller to your PC. Each of these is referred to as a “system” throughout this document. Most function calls require a system index as parameter to address a specific system. The system indexes are generated by the API and must be saved within the application.

Each system has a maximum number of “channels”. Channels are divided into two types: positioner channels and end effector channels. Each channel can control a single positioner or end effector, depending on its type. Function calls that are directed to a specific channel require a system index and a channel index to address the selected channel. The channel indexes are zero based. Note that the number of channels is constant for a given system and describes the number of positioners and/or end effectors that *may* be connected to the system and **not** the number that currently *are* connected to the system.

Please note that all functions of the library use the `cdecl` calling convention. Some development environments, such as Delphi, use `stdcall` by default. This must be taken into account when importing the library functions.

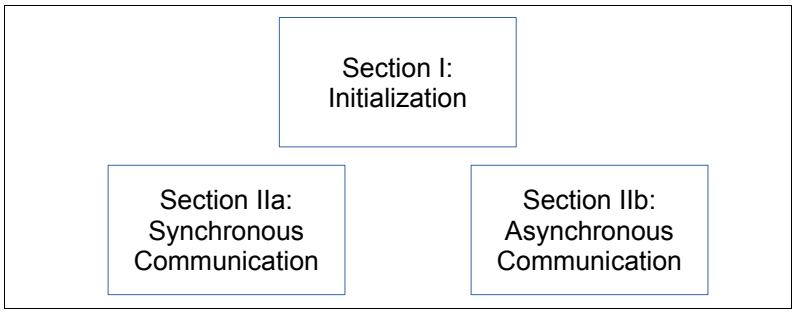
2 Functional Documentation

Besides basic features like moving and stopping positioners the MCS offers many features that are activated or configured via additional function calls. You may refer to the Detailed Function Description on how to call these functions, parameter ranges etc. However, some features require more information for a better understanding. This chapter explains these features in more detail and describes some basic concepts of the library and the MCS controller.

2.1 Overview

The functions of the library are grouped into sections. Section I is for initialization, while the functions of section II are for the actual communication with the hardware.

Section II is split up into two parts (IIa and IIb) which must be used mutually exclusive. During initialization you must choose between *synchronous communication* or *asynchronous communication*. Depending on this choice, only the functions of section IIa or IIb may be called, otherwise an error is returned.



For better distinction the function names of section IIa have a suffix of _S (synchronous) while the function names of section IIb have suffix of _A (asynchronous).

All functions of the library return a status code of type `SA_STATUS`. The return value indicates if the call was successful (`SA_OK`) or if an error occurred. See appendix 5.1 “Function Status Codes“ for a complete list of error codes. It is advised to check the return status of each function call. Some functions have output values (when retrieving information from the hardware). Their values are undefined if the return status of the function is not `SA_OK`. This may result in unwanted behavior if these values are further processed.

The `SA_GetStatusInfo` function may be used to translate a status code into a human readable text string in case you wish to output the error to the application user.

For simplicity, all function parameters are 32 bits wide, either signed or unsigned. Note though that most parameters have a limited valid range. See section 3 “Detailed Function Description“ for more information.

Note that most functions of section II are only callable for certain channel types. Please refer to the detailed function description.

2.2 Initialization

Before being able to use a system it must be initialized with a call to `SA_OpenSystem`. This function connects to the system specified in the *locator* parameter and returns a *systemIndex* (handle) to the system, if the call was successful. The returned *systemIndex* must be saved within the application and passed as a parameter to the API functions. When a connection is established you can use the functions of section II to interact with the connected system(s).

A system 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 system by calling `SA_CloseSystem` before it is free to be used by other applications. Not closing a system will cause a *resource leak*.

The older initialization functions `SA_AddSystemToInitSystemsList`, `SA_ClearInitSystemsList`, `SA_InitSystems`, `SA_GetSystemID`, `SA_GetAvailableSystems`, `SA_GetNumberOfSystems`, and `SA_ReleaseSystems` are still available in the API version 2.0 but are **deprecated**. We recommend to use only the locator-based functions `SA_OpenSystem`, `SA_CloseSystem`, `SA_FindSystems` and `SA_GetSystemLocator` for systems management.

2.2.1 Locators for System Identification

Systems are identified with *locator* strings, similar to URLs used to locate web pages. Typical locators are:

```
usb:id:3118167233
network:192.168.1.100:5000
```

The first locator identifies an MCS with the given system ID connected over USB. The second one identifies an MCS that is connected to the network.

USB Device Locator Syntax

MCS devices with USB interface can be addressed with the following locator syntax:

```
usb:id:<id>
```

where `<id>` is the first part of a USB devices serial number which is usually printed on the MCS controller. MCS with a USB interface can also be addressed with the alternative locator syntax:

```
usb:ix:<n>
```

where the number `<n>` selects the *nth* device in the list of all currently attached MCS with a USB interface. The drawback of identifying an MCS with this method is, that the number and the order of connected MCS can 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 MCS connected to the PC. We recommend to use the `usb:id:...` format for USB systems.

For USB devices you can also use the function `SA_FindSystems`, which will scan the USB ports for MCS systems and return a list with the locator strings. If you are already connected to a system you can use the function `SA_GetSystemLocator(systemIndex)` to get the locator of the system with the given index.

Network Device Locator Syntax

MCS devices with a network interface are addressed with the following locator syntax:

```
network:<ip>:<port>
```

`<ip>` is an IPv4 address which consists of four integer numbers between 0 and 255 separated by a dot. `<port>` is an integer number. For example, the locator `network:192.168.1.200:5000` addresses a device with the IP address 192.168.1.200 and TCP port 5000.

Note: 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. Ensure that all timeout parameters in calls to functions like `SA_ReceiveNextPacket_A` are adequate for the network environment.

2.2.2 Communication Modes

When calling `SA_OpenSystem` you must chose between two communication modes. The different modes affect internal communication but also the way you must use the library. The synchronous communication mode is simpler, but also less flexible. The asynchronous mode is more flexible, but requires a bit more of programming overhead.

Generally, in the synchronous mode a function call sends a command to the hardware and blocks until a response has been received. This may be the desired information, an error code or a simple acknowledge. Every command sent results in exactly one answer and the library handles the answer retrieval. The blocking time is only a few milliseconds and is therefore typically not “visible” to the programmer.

In contrast, in the asynchronous mode a function call sends a command to the hardware and returns immediately. It is then the users responsibility to fetch the answer from the hardware. A command sent may result in zero or more answers, depending on the command and the current status of the channel. Please refer to section 2.3 “Using the Asynchronous Mode” for more information.

2.3 Using the Asynchronous Mode

This section is meant to give you a better understanding of the asynchronous communication mode. It is designed to be used in several different ways in order to fit your needs.

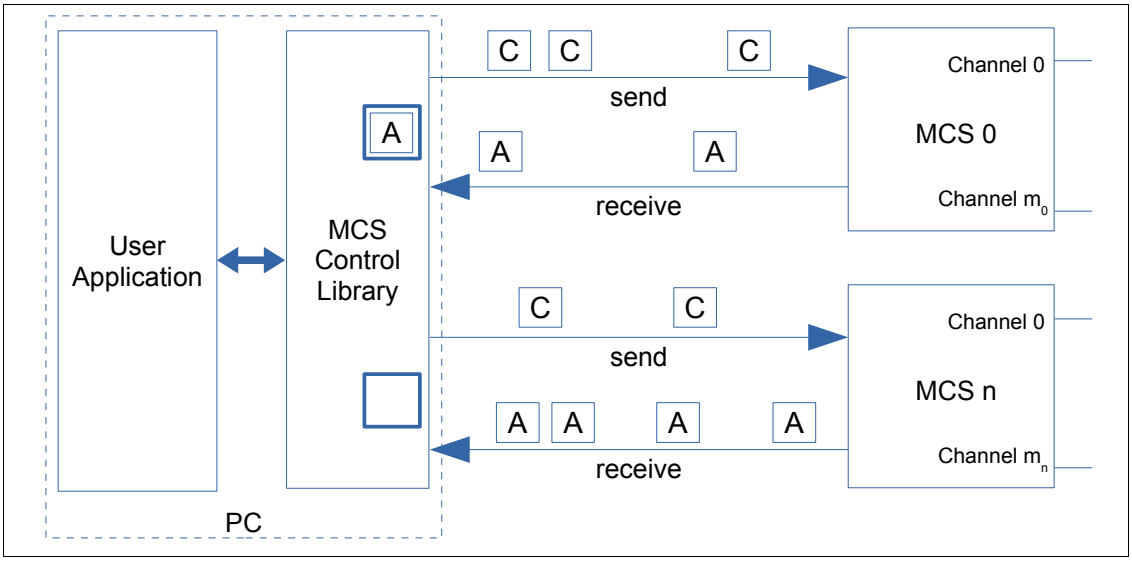
2.3.1 Overview

Generally, you can think of two communication lines that connect each MCS to the PC. One line is for transmitting commands to the MCS, the other for receiving answers from the MCS. In the asynchronous communication mode the traffic on one line is independent from the traffic on the other and the two lines need not be in sync. Thus, when a command is sent to the hardware there might be an answer or not, depending on what the command was or how the targeted channel is configured etc. Note that if several commands are sent to different channels of a system, the order of answers (if any) is not defined. However, the order of answers to commands sent to a single channel is defined by the order in which the commands were issued.

To summarize, the usage of the asynchronous mode consists of two parts:

- Sending commands: These include movement commands, configuration commands etc.
- Retrieving and managing answers: These include status information, errors etc.

The figure below gives a structural overview. Although most users will likely have only one MCS system connected to the PC, the library supports communicating with several systems in parallel.



2.3.2 Sending Commands

Most functions of section IIb of the library transmit a command to the hardware invoking some functionality. The functions do not block, but return immediately. Note that even functions like `SA_GetStatus_A` return immediately and do not provide the desired information right away. They only send a request to the hardware and the answer has to be retrieved by the user (see below).

Error handling is done on two levels. The library does some valid range checks and the like before actually sending the command. An error detected on this level will result in a status code returned by the function other than `SA_OK`.

However, even if the transmission of a command is successful, the hardware might have further restrictions or other error situations may occur. In this case the hardware will answer with an error packet. It is also the responsibility of the user to retrieve the error packet and handle the situation properly.

2.3.3 Retrieving Answers

While sending commands is straight forward (simply call a command function such as `SA_StepMove_A` and pass the desired parameters) the reception of data involves a bit more overhead. The library offers several ways of handling answer retrieval.

Generally, answers coming from an MCS are put into a receive buffer (see figure above). The library offers access to this buffer and data packets must be retrieved by the user application in the order of their arrival. Each MCS system has its own receive buffer. Thus, all answers from all channels of a given system are sequentially put into the same buffer.

The receive buffer is organized as a FIFO buffer. The following functions are used to access the head of this buffer:

- `SA_ReceiveNextPacket_A`: This is the standard answer retrieval function. It returns the answer packet that is currently located in the head of the receive buffer. If the buffer is empty and no timeout is given (timeout = 0), then a packet of type `SA_NO_PACKET_TYPE` is returned. If the buffer is empty and a timeout is given, then the first packet to arrive within the specified interval will be returned. If the time elapses before a packet arrives, a packet of type `SA_NO_PACKET_TYPE` is returned. Packets that are returned by this function are consumed, thus, removed from the packet buffer.
- `SA_LookAtNextPacket_A`: This function behaves the same way as `SA_ReceiveNextPacket_A`, with the difference that it does not consume returned packets. Packets that are returned by this function remain the receive buffer.
- `SA_DiscardPacket_A`: This function consumes a packet in the receive buffer, thus removing it. If the receive buffer is empty, the function has no effect.

Some implications:

- “Looking at” a packet stops the data flow, since the packet is not removed from the queue and you may only look at packets at the head of the queue. In order to receive further packets, call `SA_ReceiveNextPacket_A` or `SA_DiscardPacket_A`.
- Calling `SA_LookAtNextPacket_A` and then `SA_DiscardPacket_A` has the same effect as calling `SA_ReceiveNextPacket_A`. The difference is that the application (threads) may look at a packet as often as desired before it is consumed (potentially by the thread responsible for processing the specific packet).

When retrieving answers e.g. with `SA_ReceiveNextPacket_A` you can control the blocking behavior of the function via the timeout parameter. Generally, the function will return on one of two events:

- a packet is received or
- a timeout occurred.

When given a long or even infinite timeout (`SA_TIMEOUT_INFINITE`) and no packet is incoming you may cancel the function call (unblock it) by calling `SA_CancelWaitForPacket_A` from another application thread. This is typically useful when the application is to be terminated and the receiving thread must be unblocked for a proper cleanup.

Data Packet Format

A data packet is defined by the following structure:

```
typedef struct SA_packet {
       SA_PACKET_TYPE packetType; // type of packet
       SA_INDEX channelIndex;     // source channel
       unsigned int data1;        // data field
       signed int data2;          // data field
       signed int data3;          // data field
       unsigned int data4;        // data field
} SA_PACKET;
```

Any data packet that is returned by the packet retrieval functions should first be checked for its type. Whether the other fields of the data packet are valid or not depends on the type field. See appendix 5.2 “Packet Types“ for a list of packet types and their meanings.

2.3.4 Other Issues

Report on Complete

The library function `SA_SetReportOnComplete_A` configures a channel to notify the software if a movement command has been completed. If configured so and a movement has completed, the channel will send a packet of type `SA_COMPLETED_PACKET_TYPE`. For clarification, the situations in which these packets are sent are identified in the following.

The current positioner status is described by several states (see section Channel Status Codes for a list of status codes). At any given moment the positioner is in one of these states. Generally speaking, a "completed" notification is generated when entering the `SA_STOPPED_STATUS` or the `SA_HOLDING_STATUS` state. This will be the case in the following situations:

- A movement command has completed normally. These commands include `SA_StepMove_A`, `SA_ScanMoveAbsolute_A`, `SA_ScanMoveRelative_A`, `SA_GotoPositionAbsolute_A`, `SA_GotoPositionRelative_A`, `SA_GotoAngleAbsolute_A`, `SA_GotoAngleRelative_A`, `SA_CalibrateSensor_A`, `SA_FindReferenceMark_A`, `SA_GotoGripperOpeningAbsolute_A`, `SA_GotoGripperOpeningRelative_A`, `SA_GotoGripperForceAbsolute_A and SA_SetZeroForce_A`
- One of the above movement commands was aborted by an `SA_Stop_A` command, forcing the positioner into the `SA_STOPPED_STATUS` state. Note that the stop command itself does not trigger the notification. If the positioner is already in the stopped state, the stop command will have no effect.
  The only exception to this is when the positioner is in the `SA_MOVE_DELAY_STATUS` state. In this case a stop command will not trigger a notification.

Note: Closed-loop commands (e.g. `SA_GotoPositionAbsolute_A`) are considered completed when the target position has been reached and not when the optional hold time has elapsed.

Note: Overwriting movement commands (sending movement commands before the completed notification 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 completed notification is generated (when the second command completes). However, if the second command arrives just *after* the first has completed, two completed notifications are generated (one for each command).

Report on Triggered

Similar to `SA_SetReportOnComplete_A`, the library function `SA_SetReportOnTriggered_A` configures a channel to notify the software if a movement command from the command queue (see section 2.6.5 "Command Queues") has been triggered. If configured so and a movement has been triggered, the channel will send a packet of type `SA_TRIGGERED_PACKET_TYPE`.

Multiple Command Sources

It is possible to control an MCS by software while also having a Hand Control Module attached to it. A setup like this has advantages, but also disadvantages and there are several things to consider.

All channels of a system will accept commands coming either from the PC software or from the Hand Control Module. This makes it possible to have an automated software control system running on the PC while still being able to perform manual position adjustments, e.g. when the software is inactive.

However, in this situation it is also possible that commands coming from the software are overridden by the Hand Control Module and vice versa. This may result in unexpected behavior and it is the users responsibility to take this into account when writing the software or providing manual command input while the software is in control.

For the software running on the PC it is possible to detect such situations when using the asynchronous communication mode. It will receive an error packet from the affected channel with the error code `SA_COMMAND_OVERRIDEN_ERROR`.

Here is an example situation:

The software, using the asynchronous mode, configures a channel to report movement completion. It sends a command to the channel to execute an absolute position movement to the zero position and waits for the “complete” notification. While the positioner is in motion the user manually halts the movement with the Hand Control Module. The software will not receive the expected “completed” notification, but rather an error informing about the interruption.

To avoid situations like these the software may (temporarily) disable the Hand Control Module entirely. See the `SA_SetHCMEnabled` function.

2.4 Channel Properties

Each channel of an MCS controller has various properties that affect the behavior of the channel. These can be global parameters, operation modes etc. To manipulate these properties you may use the functions `SA_GetChannelProperty_S` and `SA_SetChannelProperty_S` (resp. their _A variant when using the asynchronous communication mode). When calling these functions you must supply a property key to indicate which property you wish to read or write. A property key is a 32-bit code that refers to a property and has the following structure:

Key							
31	24	23	16	15	8	7	0
component		sub component		property			

An MCS channel contains several functional components that may be further divided into several sub components. Each of these (sub) components may have several properties. The header file of the library lists definitions for various components and properties. Note that not all components may be combined with all properties when generating property keys.

While the property keys may be generated manually it is recommended to use the `SA_EPK` helper function to encode valid keys. Simply pass the desired component, sub component and property to the function and feed the result to `SA_GetChannelProperty_S` or `SA_SetChannelProperty_S`.

For example, to configure the digital input with index 0 to generate events on rising edges use the following code:

```
SA_SetChannelProperty_S(
      mcsHandle,                                  // system handle
      0,                                          // channel index
      SA_EPK(SA_DIGITAL_IN,0,SA_ACTIVE_EDGE),     // property key
      SA_RISING_EDGE                              // property value
);
```

Please refer to the appendix (5.5 “Channel Properties“) for a list of channel properties.

Some properties are described in the following sections. Properties which are related to the Controller Event System, e.g. Capture Buffers are described in section 2.6 .

2.4.1 Emergency Stop

Note: This feature is not available on all MCS controllers. Please contact SmarAct for more information.

The MCS controller may be equipped with a dedicated hardware TTL input signal that is used as an emergency stop. A negative pulse on this line (*ES line*) may cause channels of the system to stop

immediately. The operation mode property of the Emergency Stop component controls the behavior of a channel in case of such an emergency.

There are four modes available:

- SA_ESM_NORMAL: This is the default mode. In this mode a falling edge on the ES line has the same effect as issuing a single SA_Stop_S command. After such an event the system continues to behave normally.
- SA_ESM_RESTRICTED: In this mode a falling edge on the ES line will issue a stop and make the channel enter a locked state. In this state you may communicate with the channel normally, but all movement commands will return an SA_MOVEMENT_LOCKED_ERROR. The locked state may be reset by setting the operation mode to any valid value, thereby unlocking the movement again.
- SA_ESM_DISABLED: In this mode falling edges on the ES line are simply ignored.
- SA_ESM_AUTO_RELEASE: In this mode a falling edge on the ES line will issue a stop and make the channel enter a locked state. In this state you may communicate with the channel normally, but all movement commands will return an SA_MOVEMENT_LOCKED_ERROR. This state remains until the channel detects a rising edge on the ES line or the operation mode is set to any valid value.

Note that the ES line will be also triggered internally when the USB cable is unplugged.

The default mode of the Emergency Stop feature can be changed with the Channel Property SA_DEFAULT_OPERATION_MODE.

Code example:

```
// disable the ES line
SA_STATUS result = SA_SetChannelProperty_S(
      mcsHandle,                                                  // system handle
      0,                                                          // channel index
      SA_EPK(SA_GENERAL,SA_EMERGENCY_STOP,SA_OPERATION_MODE),     // property key
      SA_ESM_DISABLED                                             // property value
);
```

2.4.2 Low Vibration

SmarAct's positioners allow macroscopic movement while still offering ultra-high precision on the nanometer scale. However, the stick-slip driving principle is accompanied by high-frequent vibrations that may cause trouble in some applications.

For this the MCS offers a special operation mode in which movement commands are executed to produce as little vibrations as possible. To activate this mode simply set its operation mode property to SA_ENABLED.

```
SA_STATUS result = SA_SetChannelProperty_S(
      mcsHandle,                                                  // system handle
      0,                                                          // channel index
      SA_EPK(SA_GENERAL,SA_LOW_VIBRATION,SA_OPERATION_MODE),      // property key
      SA_ENABLED                                                  // property value
);
```

Note that the Low Vibration mode requires the acceleration control feature to be active (see SA_SetClosedLoopMoveAcceleration_S). Enabling the Low Vibration mode while acceleration control is inactive will cause the acceleration control to be implicitly activated with a default value.

Also note that a channel must be completely stopped (SA_STOPPED_STATUS) in order to be able to change the Low Vibration operation mode. See also section 2.7.2 “Dependency Chains“.

Note: This feature is not available on all controllers. Please contact SmarAct for more information.

2.4.3 Broadcast Stop

This feature can trigger an Emergency Stop on the MCS controller when an end stop is detected. It is typically useful when multiple channels are moving simultaneously and an end stop on one channel should cause a halt on all other channels. Please note that a channel whose Emergency Stop Mode is set to `SA_ESM_DISABLED` is not affected by this trigger.

```
// trigger an emergency stop when an end stop on channel 0 occurred
SA_STATUS result = SA_SetChannelProperty_S(
      mcsHandle,                                                  // system handle
      0,                                                          // channel index
      SA_EPK(SA_GENERAL,SA_BROADCAST_STOP,SA_OPERATION_MODE),     // property key
      SA_ENABLED                                                  // property value
);
```

2.4.4 Position Control

This sub component is used to change parameters which are associated with a channels movement controller. Currently there is only one property available.

Forced Slip: When reaching a target position, e.g. after a `SA_GotoPositionAbsolute_S` was issued, the channel will try to stop at approx. 50% of its step size, thus improving the holding feature. If this behavior is unwanted it can be disabled with this channel property. This feature is enabled by default.

```
// deactivate the Forced Slip feature
SA_STATUS result = SA_SetChannelProperty_S(
      mcsHandle,                                                  // system handle
      0,                                                          // channel index
      SA_EPK(SA_GENERAL,SA_POSITION_CONTROL,SA_FORCED_SLIP),      // property key
      SA_DISABLED                                                 // property value
);
```

2.4.5 Sensor

This sub component controls the channel parameters which specify the handling of the sensor. Please note that this features are not available on channels without sensor.

Power Supply

This property selects if the sensor should be enabled, disabled or in power save mode, for more details on the operation modes refer to section 2.5.2 . Whereas the command `SA_SetSensorEnabled_S` sets the property for all channels of an MCS Controller (see p. 84), this feature can be used to modify the operation mode of a single channel.

```
// set the sensor on channel 0 in power save mode
SA_STATUS result = SA_SetChannelProperty(
      mcsHandle,                                                  // system handle
      0,                                                          // channel index
      SA_EPK(SA_SENSOR,SA_POWER_SUPPLY,SA_OPERATION_MODE),        // property key
      SA_POWERSAVE
);
```

Scale

This sub component allows an alternative access to the scale settings of the channel. For more details on this functionality refer to paragraph “Shifting the Measuring Scale“ on p.18.

```
// set a scale shift of 1mm on channel 0
SA_STATUS result = SA_SetChannelProperty(
      mcsHandle,                                                  // system handle
      0,                                                          // channel index
      SA_EPK(SA_SENSOR,SA_SCALE,SA_OFFSET),                       // property key
      1000000
);
```

2.5 Working With Sensor Feedback

This section covers some features of the MCS when using positioners with integrated sensors and explains them in more detail.

2.5.1 Rotary Sensors

In contrast to linear sensors, where the position is simply given by a single signed value, rotary sensors are handled a little differently.

Suppose a rotary positioner is currently aligned to a 45° angle and shall be instructed to move to 90°. This can be accomplished in two ways: clockwise or counter-clockwise. To eliminate such ambiguities, a rotary position is defined by a combination of an angle and a revolution. The angle value is given in micro degrees and has a valid range of 0..359,999,999. The revolution value indicates complete 360° rotations of the positioner and has a valid range of -32,768..32,767.

If a rotary positioner starting at zero (angle = 0; revolution = 0) moves in positive direction, the angle value will increase, reflecting the current orientation of the positioner. If the positioner moves further over the 360° boundary, the angle value will wrap around to zero and the revolution value will be incremented by 1 to indicate one full rotation of the positioner. The reverse direction is done accordingly.

Consequently, when issuing absolute movement commands with `SA_GotoAngleRelative_S`, the movement direction is implicitly defined by the parameters given. In the example above (moving from 45° to 90°) the direction would be distinguished by specifying 0 or -1 as the revolution parameter (assuming the current revolution is 0).

Note that the valid range of `SA_GotoAngleRelative_S` is extended in the negative range to -359,999,999..359,999,999. This is simply for convenience. For example, the following commands have the same effect:

```
SA_GotoAngleRelative_S(mcsHandle,0,-90000000,0,0);
SA_GotoAngleRelative_S(mcsHandle,0,270000000,-1,0);
```

Both commands will move the positioner 90° in negative direction.

2.5.2 Sensor 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 with the `SA_SetSensorEnabled_S` function.

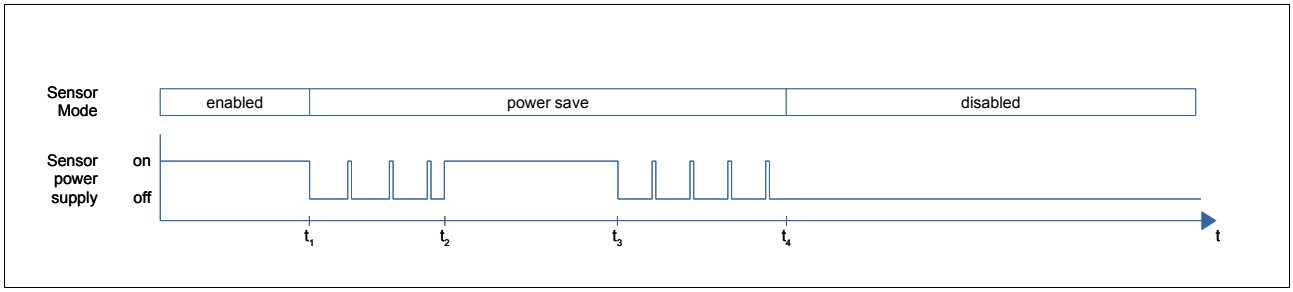
- *Disabled* - In this mode the power supply of the sensor is turned off. This avoids the generation of heat. Closed-loop commands such as `SA_GotoPositionAbsolute_S` will not be executed, but rather an error returned informing about the sensor state. This mode may also be useful if the light that is emitted by the sensors interferes with other components of your setup (e.g. detectors inside an SEM chamber).

Note: Open-loop commands such as `SA_StepMove_S` are still executed. This implies that the position information will become invalid, since the positioner cannot track its position during the movement. It is the users responsibility to enable the sensors again before moving the positioner, should the position tracking be needed. Be aware that position calculation is done incrementally. So even after turning on the sensors again the position will be invalid if the positioner was moved while the sensor was offline.

- *Enabled* - In this mode the sensor is supplied with power continuously. All movement commands are executed normally.
- *Power Save* - If set to this mode the power supply of the sensor will be handled by the system 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 system 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 during this time.

The figure below illustrates the different sensor modes and shows when the sensors are supplied with power.



In this example the sensor mode is initially set to *enabled*. The sensors are continuously supplied with power. At time t_1 the sensor mode is switched to *power save*. In this mode the system starts to pulse the power supply of the sensors to keep the heat generation low. At time t_2 a movement command is issued, which requires the sensors to be online in order to keep track of the current position. Note that the sensor mode stays unchanged during this time. As soon as the movement has finished (t_3) the system will start to pulse the power supply again. At time t_4 the sensor mode is switched to *disabled*, in which the power supply is turned off continuously. If movement commands are issued during this time then the system will not be able to track the position. The position data will become invalid.

Notes on the power save mode:

If closed-loop commands are issued with a hold time, then the system will start to pulse the power supply of the sensor as soon as the target position has been reached (t_3 in the above example). At this point the hold time starts. The positioner will still hold the target position and compensate for drift effects while pulsing, although it might not be as accurate as in the *enabled* mode.

When a movement command is issued in power save mode the sensors have to be temporarily enabled before the movement can begin. Powering up the sensors takes a few milliseconds. If the status of the positioner is polled during this time the status will be `SA_MOVE_DELAY_STATUS`. This is a pit fall in the following code example:

```
SA_SetSensorEnabled_S(mcsHandle,SA_SENSOR_POWERSAVE);
SA_StepMove_S(mcsHandle,0,1000,4095,1000);
unsigned int status;
SA_GetStatus_S(mcsHandle,0,&status);
while (status == SA_STEPPING_STATUS) {
	SA_GetStatus_S(mcsHandle,0,&status);
}
```

It is very likely that the while-loop terminates immediately, although the movement is not yet completed (hasn't even started). In this case it is better to use this code:

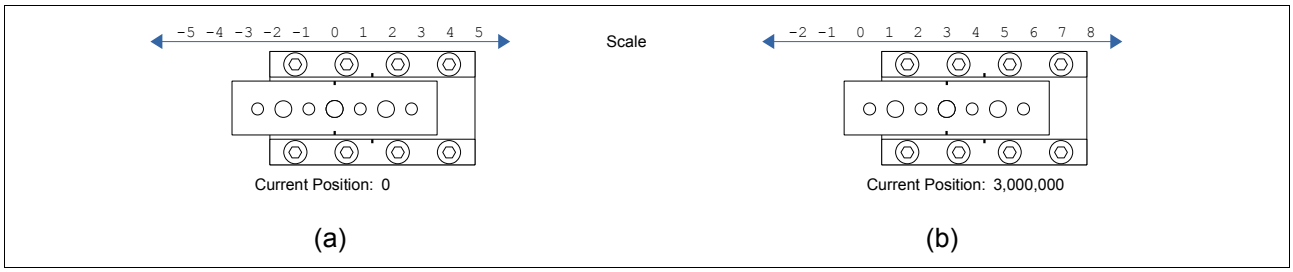
```
SA_SetSensorEnabled_S(mcsHandle,SA_SENSOR_POWERSAVE);
SA_StepMove_S(mcsHandle,0,1000,4095,1000);
unsigned int status;
SA_GetStatus_S(mcsHandle,0,&status);
while (status != SA_STOPPED_STATUS) {
	SA_GetStatus_S(mcsHandle,0,&status);
}
```

2.5.3 Defining Positions

Since position calculation is done on an incremental basis, the MCS 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 `SA_SetPosition_S` function may be used 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).

The figure below 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 a call of
`SA_SetPosition_S(mcsHandle,0,3000000);`
The current position has been defined to +3mm and the measuring scale is shifted accordingly.



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_FindReferenceMark_S` function may be used to move a positioner to a known 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 Shifting the Measuring Scale.

Depending on the type of your positioner it may be equipped with a single reference mark or with multiple reference marks. Some positioners do not have a physical reference mark, but are rather referenced via a mechanical end stop. The different search algorithms are outlined in the following:

- Single Reference Mark: In this case the reference mark (which is usually located near the middle of the travel range) is used to determine the physical position. The positioner starts to move in the given initial direction. As soon as the reference mark has been detected the positioner stops and the search is successful. If the positioner detects an end stop then the search direction is reversed and the search is continued. If a second end stop is detected before the reference mark is found the search will abort unsuccessfully. (When using the asynchronous communication mode an error will be generated.)
  For these types of positioners (linear, but also rotary) the physical measuring scale is defined such that the zero position is located on the reference mark.
- Distance Coded Reference Marks: In this case the distance between any two neighboring reference marks is measured in order to determine the physical position. The positioner starts to move in the given initial direction. When the first reference mark has been detected the current position value is stored and the search continues for a second mark - either in the same direction (when called with `SA_FORWARD_DIRECTION` or `SA_BACKWARD_DIRECTION`) or in reverse direction (when called with `SA_FORWARD_BACKWARD_DIRECTION` or `SA_BACKWARD_FORWARD_DIRECTION`). When the second reference mark has been detected then the positioner stops and the search is successful. The distance between the two reference marks is calculated to determine the physical position. Should the positioner detect an end stop then the search direction is reversed and the process is repeated. If a second end stop is detected before two reference marks have been found the search will abort unsuccessfully. (When using the asynchronous communication mode an error will be generated.)

For these types of positioners the physical measuring scale is defined such that half the length of the positioner is near the middle position of the slider. (The physical zero position is unreachable, since it lies outside the travel range.) For example, an SLC1730sc positioner that has its slider located at the middle position will have a physical position of 15mm. Since it has a travel range of about 22mm, the position range will be about 4mm .. 26mm.

- End Stops: In this case a mechanical end stop is used as a known physical position. The positioner will move in the safe direction (see `SA_SetSafeDirection_S`) until it detects an end stop. The sensor signals are then used to align the position to the reference position with high repeat accuracy. Note that the configured end stop must be calibrated with `SA_CalibrateSensor_S` before it can be properly used as a reference point.
  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 setting.

When the “find reference” command has completed successfully the system knows the physical position of the positioner (see also `SA_GetPhysicalPositionKnown_S`).

Shifting the Measuring Scale

The *physical* measuring scale is fix for each positioner (see previous section) and cannot be changed. However, the MCS controller uses a *logical* measuring scale when calculating positions to result in position values returned by `SA_GetPosition_S` (or `SA_GetAngle_S`). The logical measuring scale may be shifted or inverted by the user so that the controller returns a desired 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:

- Calling `SA_SetPosition_S` sets the offset implicitly by shifting the logical scale so that the current position equals the desired value (see example below).
- Calling `SA_SetScale_S` sets the offset and inversion explicitly and the current position will have a value that reflects the new scale shift and direction.

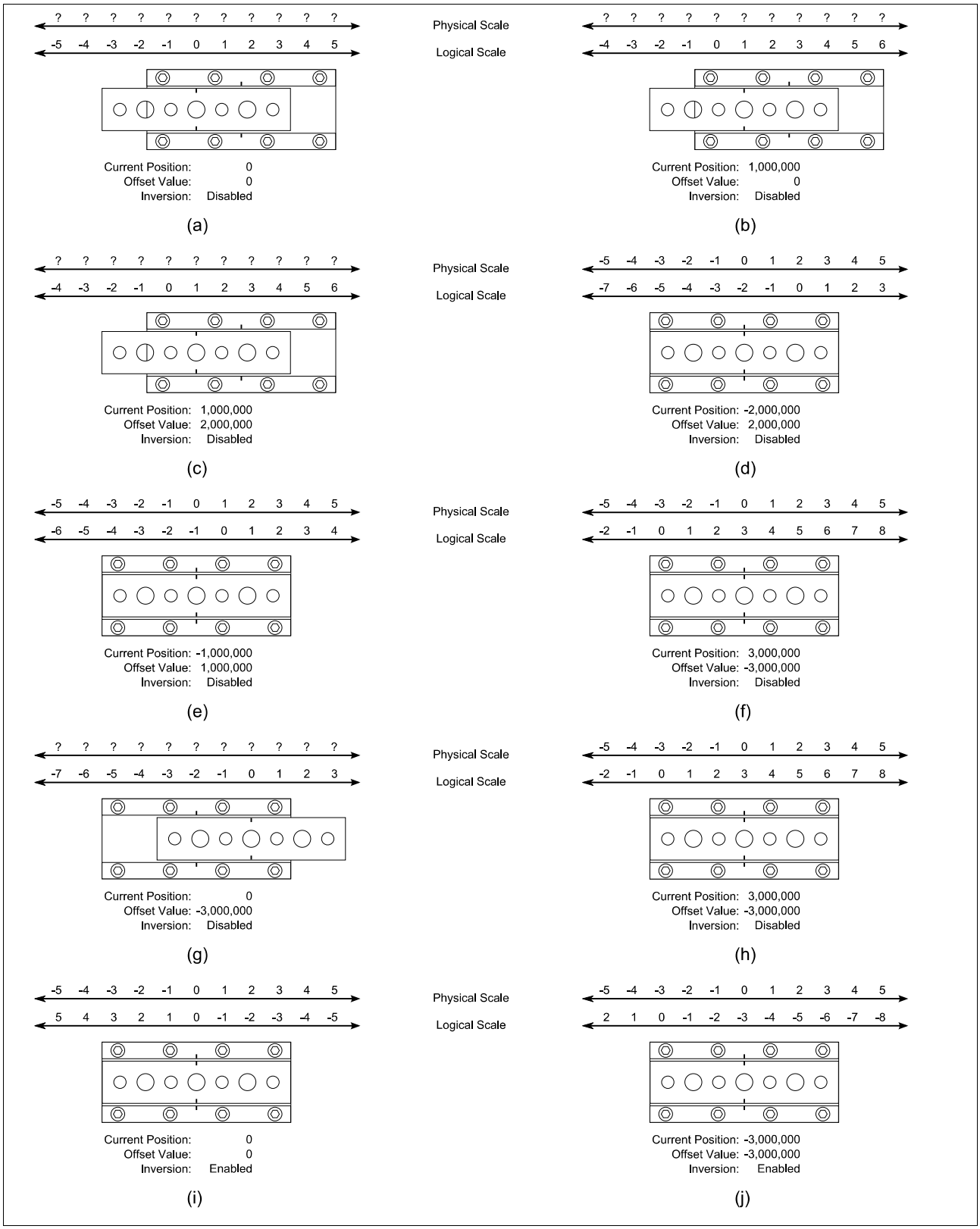
The offset and inversion value is stored in non-volatile memory. Once it is configured you only need to call `SA_FindReferenceMark_S` to restore your settings on future power-ups.

Note: The behavior of the system when calling `SA_SetPosition_S` or `SA_SetScale_S` differs slightly depending on whether the physical position is known or not (see `SA_GetPhysicalPositionKnown_S`). When the physical position is unknown a call to `SA_SetPosition_S` will not update the offset value in the non-volatile memory. Likewise, a call to `SA_SetScale_S` will have no immediate effect on the values returned by `SA_GetPosition_S`. The following table summarizes the behavior.

	physical position is known		physical position is unknown	
	`SA_SetScale_S`	`SA_SetPosition_S`	`SA_SetScale_S`	`SA_SetPosition_S`
offset value is written to non-volatile memory	yes	yes	yes	no
function call has immediate effect on position values	yes	yes	no	yes

Example

To further demonstrate the behavior of the system in different situations the figure below shows an example of an SLC positioner with a single reference mark. The small markings indicate the location of the reference mark. When the markings overlap the positioner is on the mark. The code section below shows the corresponding command sequence.



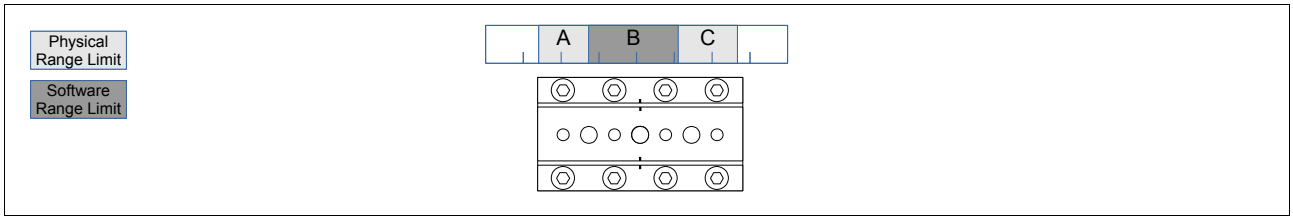
```
// (a) - The system is powered up. The physical position is unknown and the
//       current position is assumed to be 0. The offset value is in the default
//       setting.
SA_SetPosition_S(mcsHandle,0,1000000);
// (b) - The current position has been set to +1mm. Since the physical position is
//       unknown, the offset value in the non-volatile memory could not be updated
//       implicitly.
SA_SetScale_S(mcsHandle,0,2000000, SA_FALSE);
// (c) - The offset value in the non-volatile memory has been set to +2mm. Since
//       the physical position is unknown, the current position could not
//       be updated implicitly.
SA_FindReferenceMark_S(mcsHandle,0,SA_FORWARD_DIRECTION,60000,SA_NO_AUTO_ZERO);
// (d) - The positioner has moved to the reference mark. The physical position
//       is now known and the position value has been updated to reflect the
//       configured offset between the physical and the logical scale.
SA_SetPosition_S(mcsHandle,0,-1000000);
// (e) - The current position has been set to -1mm. Since the physical position is
//       known, the offset value in the non-volatile memory was updated implicitly.
SA_SetScale_S(mcsHandle,0,-3000000, SA_FALSE);
// (f) - The offset value in the non-volatile memory has been set to -3mm. Since
//       the physical position is known, the current position was updated implicitly.
// (g) - The system was shut down, the positioner was moved externally to some
//       random location and the system is then powered up again. The physical
//       position is unknown and the positioner assumes its current position as 0
//       again.
SA_FindReferenceMark_S(mcsHandle,0,SA_BACKWARD_DIRECTION,60000,SA_NO_AUTO_ZERO);
// (h) - The positioner has moved to the reference mark. The physical position
//       is now known and the position value has been updated to reflect the
//       configured offset between the physical and the logical scale.
SA_SetScale_S(mcsHandle,0,0,SA_TRUE);
// (i) - The offset and inversion value in the non-volatile memory have been set to 0mm
//       and SA_TRUE. Thus the counting direction of the scale got inverted. Since
//       the physical position is known, the current position was updated implicitly.
SA_SetScale_S(mcsHandle,0,-3000000,SA_TRUE);
// (j) - The offset value was changed and both values were stored inside the non-volatile
//       memory. Since the physical position is known, the current position is updated
//       implicitly. Notice the difference between (i) and (h).
```

2.5.4 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 MCS offers to limit the travel range of a positioner by software. The functions `SA_SetPositionLimit_S`, `SA_GetPositionLimit_S`, `SA_SetAngleLimit_S` and `SA_GetAngleLimit_S` offer control over this feature.

By default no range limit is set. Once it is defined the positioner will not move beyond the boundaries of the range limit. This affects all movements except scan movements (e.g. `SA_ScanMoveAbsolute_S`). If a movement command is issued that would move the positioner beyond the defined limit then the positioner is stopped. (When using the asynchronous communication mode an error will be generated.) 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. The figure below shows an example of a linear positioner. The positioner is limited by software to move only within zone B. Should the positioner somehow be pushed into zone A it will only be allowed to move to the right towards zone B. The same applies to zone C accordingly.

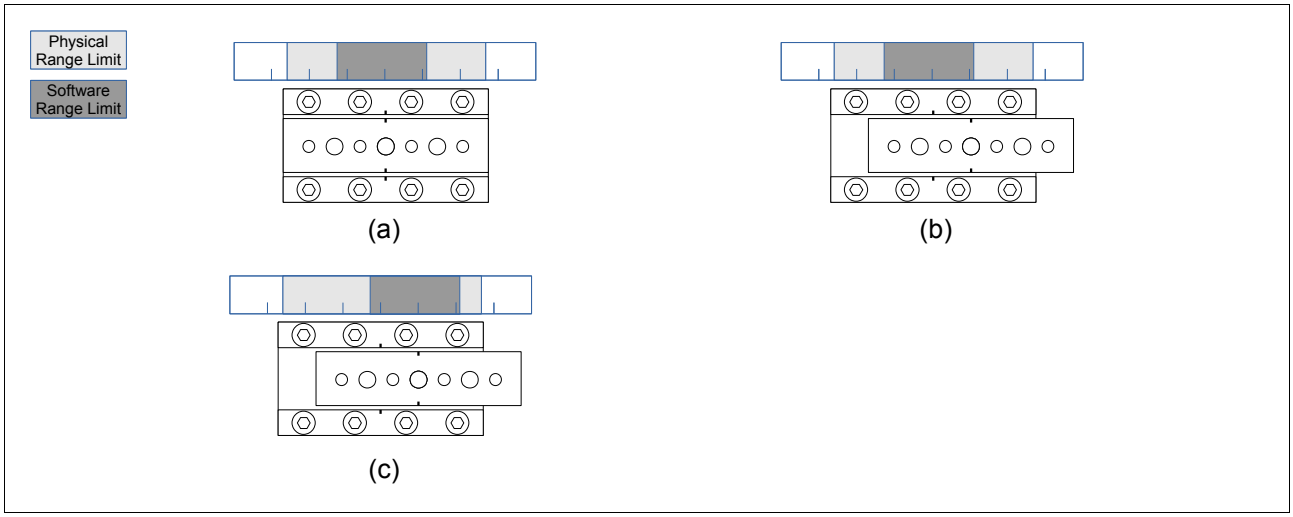


Please note the following restrictions:

- The range limit may only be set if the positioner “knows” its physical position, i.e. after the reference mark has been found (see `SA_FindReferenceMark_S`). The function `SA_GetPhysicalPositionKnown_S` may be used to check this special state.
- The range limit is *not* saved to non-volatile memory and must be configured in each session. Typically, after a system power-up you would call `SA_FindReferenceMark_S` and then `SA_SetPositionLimit_S`.
- 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.

The software range limit has some consequences that you might want to consider.

- Both the minimum and maximum position of the range limit behave similarly to a physical end stop. For example, the `SA_FindReferenceMark_S` 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.
  You should also avoid calling `SA_CalibrateSensor_S` while near a range limit boundary. Otherwise the calibration will be aborted.
- Calling `SA_SetPosition_S` does **not** automatically adjust the software range limit accordingly. (The same applies to `SA_SetScale_S`.) This means that shifting the measurement scale of the positioner with these commands will also shift the physical position of the software range limit. This is illustrated in the figure below. Suppose the positioner is currently at position 0. A software range limit is defined as indicated by the dark gray area in (a). In (b) the positioner has moved one unit to the right. After this the current position is set to zero again (with `SA_SetPosition_S`) as shown in (c). As a result, the physical position of the software range limit has moved to another location, which enables the positioner to move beyond the boundary that was defined in (a). Therefore, care should be taken when working with these commands.



2.6 Controller Event System

Each positioner channel of an MCS includes an event system that may be configured flexibly to trigger various internal features. Generally you can think of components that can generate events (event sources) and components that may be configured to receive events (event receivers). The latter ones may trigger actions when an event occurs. The table below lists the currently available event sources and event receivers.

Component	Event Source	Event Receiver
Digital In	x	
Software Trigger	x	
Counter		x
Capture Buffer		x
Command Queue		x

Event sources and receivers must be configured with the `SA_SetChannelProperty_S` function (see also section 2.4 “Channel Properties“). Event sources must be enabled (as they are disabled by default) and usually given a parameter to tell them under which conditions to generate an event. Event receivers must be “connected” to an event source (by default no source is configured) to trigger its functionality.

Note that it is insufficient to only configure an event source. If no receiver component is configured to receive the events, then the functionality of the receiver component will not trigger.

To “connect” an event receiver component with an event source you must set its trigger source property. The value of a trigger source property is a **selector value** which is (similar to property keys) a 32-bit code that refers to an event source and has the following structure:

Selector value							
31	24	23	16	15	8	7	0
unused				component		index	

While the selector values may be generated manually it is recommended to use the `SA_ESV` helper function to encode valid selectors. Simply pass the desired component and index to the function and feed the result to `SA_SetChannelProperty_S`.

For example, to configure counter 0 to listen to events generated from the digital in 0 event source, use the following code:

```
SA_SetChannelProperty_S(
       mcsHandle,                                  // system handle
       0,                                          // channel index
       SA_EPK(SA_COUNTER,0,SA_TRIGGER_SOURCE),     // property key
       SA_ESV(SA_DIGITAL_IN,0)                     // selector value
);
```

To “disconnect” an event receiver component from an event source simply set its trigger source property to `SA_DISABLED`.

Note: The Command Queue cannot be configured with a global trigger source (it has no trigger source property). Rather the commands in the queue are each given an individual trigger source when calling the `SA_AppendTriggeredCommand_A` function. The *triggerSource* parameter of this function is used in the same way as a trigger source component property, except that a value of `SA_DISABLED` is not allowed.

The components that generate and receive events are described in more detail in the following sections.

2.6.1 Digital Inputs

A digital input is an external TTL input line that is connected to a channel of the MCS controller. It may be used to generate internal events which may trigger other components.

Note: This feature is not available on all MCS controllers. Please contact SmarAct for more information.

A digital input's sub component is an index value which selects a specific input. Currently there are two inputs available, so the index value must be either 0 or 1.

Digital inputs have two component properties:

- `SA_OPERATION_MODE` (Read/Write): Can be `SA_DISABLED` (default) or `SA_ENABLED`. Enables or disables event generation. If disabled, event receivers connected to this event source will not receive any events and therefore not trigger their functionality.
- `SA_ACTIVE_EDGE` (Read/Write): Can be `SA_FALLING_EDGE` (default) or `SA_RISING_EDGE`. Determines on which edge events are generated.

Note that the active edge property value has no effect if the operation mode is disabled. It is recommended to configure the active edge before enabling the input.

The table below lists the electrical specification of the digital inputs.

	Input 0		Input 1		
Parameter	***Minimum***	***Maximum***	***Minimum***	***Maximum***	***Unit***
Pulse Frequency		1000		1000	Hz
Pulse Width	0.1		100		µs
Input Low Voltage	-0.5	1.5	-0.5	1.5	V
Input High Voltage	3	5.5	3	5.5	V

2.6.2 Software Triggers

A software trigger is an event source that is triggered when the `SA_TriggerCommand_A` function is called and is typically used in combination with command queuing (see 2.6.5 “Command Queues“) to synchronize movement commands of several channels.

A software trigger's sub component is an index which may be used to distinguish between different triggers (although one trigger is sufficient in most cases, so the index is typically zero).

The software trigger has no component properties and therefore cannot be configured. It is always implicitly enabled.

2.6.3 Counters

Counters are used to count events. They may be configured with an event source to increment the counter value every time an event is received.

A counter's sub component is an index value which selects a specific counter. Currently there is only one counter available, so the index value must be 0.

Counters have two component properties:

- `SA_TRIGGER_SOURCE` (Read/Write): Selects an event source to trigger the counter. The value of this property is a selector value. See section 2.6 “Controller Event System“.
- `SA_VALUE` (Read/Write): Reading this property returns the current counter value. You may also write this property, e.g. to reset the counter to zero or set an arbitrary starting value.
  The valid range is 0 .. 2,147,486,647.

2.6.4 Capture Buffers

When reading feedback data from a channel you are normally forced to read multiple data values consecutively which generates a time gap between the queries. However, there might be situations where you wish to synchronize the data.

Capture buffers take snapshots of groups of internal values when events occur. Reading out the capture buffer contents therefore enables you to perform an atomic read operation on multiple data values (see `SA_GetCaptureBuffer_S`).

The type of data that a capture buffer holds cannot be configured. You may only choose a buffer via its index to capture predefined value combinations. When a capture buffer is queried, an `SA_PACKET` structure is returned that holds the capture buffer contents.

The following table lists the currently defined capture buffers and the data that they capture.

data1 (buffer index)	data2	data3	data4
0	position	revolution	counter 0

Once an event source is configured for a capture buffer the corresponding internal values are copied to the capture buffer if an event is received. The values in the capture buffer remain unchanged until the next event is received.

Since the content of a capture buffer is defined by its index, capture buffers have only one property:

- `SA_TRIGGER_SOURCE` (Read/Write): Selects an event source to trigger the capture buffer. The value of this property is a selector value. See section 2.6 “Controller Event System“.

2.6.5 Command Queues

Normally when a channel receives a movement command it will be executed immediately. However, it is possible to let the execution be delayed until a certain event occurs. This allows for example to synchronize movements with external processes.

Note: This feature is only available in the asynchronous communication mode.

The command queue is organized as a FIFO queue, i.e. the first command that was appended to the queue is the first one to be executed as soon as the configured event occurs.

To append a command to the command queue you must issue two function calls consecutively:

1. Call `SA_AppendTriggeredCommand_A` and pass an event source that should trigger the command. The event source is coded in form of a selector value (see section 2.6 “Controller Event System“).
2. Call the desired movement command, e.g. `SA_GotoPositionAbsolute_A`. Note that calling a non-movement command here will generate an `SA_COMMAND_NOT_TRIGGERABLE_ERROR`.

The movement command will not be executed immediately, but rather stored in the command queue and executed as soon as it is triggered by the configured event source.

Please note that there is no way to read out the command queue. Therefore, you should keep track of the queued commands and how they are triggered in your software application if necessary.

If there is at least one command in the command queue you will not be able to execute further movement commands until the queue is empty again. You may append more commands to the queue (if not full), but issuing movement commands without previously calling `SA_AppendTriggeredCommand_A` will generate an `SA_WAITING_FOR_TRIGGER_ERROR`. The command queue will become empty if all commands in the queue have been triggered. Alternatively, you may clear the queue manually by calling `SA_ClearTriggeredCommandQueue_A`.

A channel may be configured to send a notification when a command has been triggered by calling `SA_SetReportOnTriggered_A` (see there).

Command queues have two properties:

- SA_SIZE (Read only): Indicates how many commands are currently in the queue. The value of this property will be incremented when a command is appended to the queue and decremented when the next command in the queue is triggered. (A queued command that is triggered is removed from the queue. It does not reside in the queue while it is in execution.)
- SA_CAPACITY (Read only): Indicates how many commands the queue is able to store. This value is a constant and currently has a value of 1.

Note that the command queue has no global trigger source property. The commands in the queue each have their own trigger source.

2.6.6 Example

This section demonstrates how to configure a channel for an example scenario.

Suppose you have an external system that generates positive TTL pulses on some event. You wish to count the number of pulses that occurred. On each pulse you also wish to capture the current position of a channel along with the current pulse counter value. Furthermore, the first pulse should be synchronized with a movement of the positioner.

For this scenario you could use the following code:

```
SA_STATUS result;
const char loc[] = "usb:id:3118167233";
unsigned int mcsHandle;
// Connect to system (must be async to be able to use command queue).
result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) return result;
// Make the channel notify us when the positioner starts moving.
result = SA_SetReportOnTriggered_A(mcsHandle,0,SA_ENABLED);
if (result != SA_OK) return result;
// Make the channel notify us when the positioner completes the movement.
result = SA_SetReportOnComplete_A(mcsHandle,0,SA_ENABLED);
if (result != SA_OK) return result;
// Configure counter 0 to listen to digital in 0.
result = SA_SetChannelProperty_A(
      mcsHandle,                                    // system handle
      0,                                            // channel index
      SA_EPK(SA_COUNTER,0,SA_TRIGGER_SOURCE),       // property key
      SA_ESV(SA_DIGITAL_IN,0)                       // selector value
);
if (result != SA_OK) return result;
// Configure capture buffer 0 to listen to digital in 0.
result = SA_SetChannelProperty_A(
      mcsHandle,                                    // system handle
      0,                                            // channel index
      SA_EPK(SA_CAPTURE_BUFFER,0,SA_TRIGGER_SOURCE),// property key
      SA_ESV(SA_DIGITAL_IN,0)                       // selector value
);
if (result != SA_OK) return result;
// Configure digital in 0 for rising edges.
result = SA_SetChannelProperty_A(
      mcsHandle,                                    // system handle
      0,                                            // channel index
      SA_EPK(SA_DIGITAL_IN,0,SA_ACTIVE_EDGE),       // property key
      SA_RISING_EDGE                                // property value
);
if (result != SA_OK) return result;
// Enable digital in 0.
result = SA_SetChannelProperty_A(
      mcsHandle,                                    // system handle
      0,                                            // channel index
      SA_EPK(SA_DIGITAL_IN,0,SA_OPERATION_MODE),    // property key
      SA_ENABLED                                    // property value
```

```
);
if (result != SA_OK) return result;
// Add a movement command to the command queue.
result = SA_AppendTriggeredCommand_A(
       mcsHandle,                                    // system handle
       0,                                            // channel index
       SA_ESV(SA_DIGITAL_IN,0)                       // selector value
);
if (result != SA_OK) return result;
// The next movement command will not be executed right away,
// but fed into the command queue.
result = SA_GotoPositionAbsolute_A(mcsHandle,0,1000000,0);
if (result != SA_OK) return result;
// The channel is now “armed”. The first pulse on the digital in 0 input will cause the
// positioner to move by +1mm. Pulses will be counted and position values captured.
// Wait for the first pulse.
SA_PACKET packet;
do {
       result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
       if (result != SA_OK) return result;
       // Return an error in case one of the above commands caused an error.
       if (packet.packetType == SA_ERROR_PACKET_TYPE) return packet.data1;
} while (packet.packetType != SA_TRIGGERED_PACKET_TYPE);
// The positioner is on the move now.
// Read position data in a loop until the movement completes.
while (1) {
       // Request capture buffer data.
       result = SA_GetCaptureBuffer_A(mcsHandle,0,0);
       if (result != SA_OK) return result;
       // Read answer.
       result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
       if (result != SA_OK) return result;
       if (packet.packetType == SA_CAPTURE_BUFFER_PACKET_TYPE) {
              // Store position and counter data to some location.
              logData(packet);
       } else if (packet.packetType == SA_COMPLETED_PACKET_TYPE) {
              // Abort loop if the movement is complete.
              break;
       }
       msleep(20);  // Only poll data with about 50Hz.
}
// Done.
SA_CloseSystem(mcsHandle);
```

2.7 Miscellaneous Topics

2.7.1 Overwriting Movement Commands

Generally, the function calls for movement commands return as soon as the command has been transmitted to the hardware (and an acknowledge received in synchronous mode); 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.

However, a special situation occurs when overwriting movement commands while using the report-on-complete-feature in the asynchronous mode. See section ?? “Report on Complete” for more information.

2.7.2 Dependency Chains

Some configuration options of a channel depend on other options to be set. Specifically, there is the dependency chain

Speed Control ← Acceleration Control ← Low Vibration Mode.

The arrows represent a “requires” relation. Hence, only the following configuration combinations are valid:

State	Speed Control	Acceleration Control	Low Vibration Mode
1	inactive	inactive	inactive
2	active	inactive	inactive
3	active	active	inactive
4	active	active	active

When activating or deactivating one of these features the system implicitly takes actions to maintain a consistent state. E.g. when activating the low vibration mode while in state 1, the speed control and acceleration control features will be implicitly enabled (with default settings). Likewise, deactivating the speed control while in state 4, will implicitly deactivate the acceleration control and the low vibration mode.

Note that a channel must be completely stopped in order to change the low vibration mode. Otherwise an `SA_COMMAND_NOT_PROCESSABLE_ERROR` will be generated. This implies that deactivating the speed control may also produce this error, if the low vibration mode cannot be implicitly deactivated.

Note: The Speed Control, Acceleration Control and Low Vibration Modes are not available on all controllers. Please contact SmarAct for more information.

3 Detailed Function Description

Please note that all functions of the library use the `cdecl` calling convention. Some development environments, such as Delphi, use `stdcall` by default. This must be taken into account when importing the library functions.

3.1 Initialization Functions

SA_AddSystemToInitSystemsList

Caution: This function has been deprecated. Please use the function `SA_OpenSystem` for initialization. Can only be used in conjunction with `SA_InitSystems` . This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_AddSystemToInitSystemsList(unsigned int systemId);
```

Description:

This function may be used to acquire specific systems at initialization. The library manages a list of system IDs that are to be acquired when calling `SA_InitSystems`. Initially, this list is empty in which case *all* available systems are acquired. If you wish to acquire one or more specific systems add their system IDs to the list with this function.

Parameters:

- *systemId* (unsigned 32bit), input - ID of the system that is to be added to the system ID list.

Example:

```
SA_ClearInitSystemsList();
SA_AddSystemToInitSystemsList(487519957);
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);
if (result == SA_OK) {
	// system was successfully acquired
}
```

See also: `SA_GetAvailableSystems`, `SA_ClearInitSystemsList`, `SA_InitSystems`

SA_CloseSystem

Interface:

```
SA_STATUS SA_CloseSystem(SA_INDEX systemIndex);
```

Description:

This function closes a system initialized with `SA_OpenSystem`. It should be called before the application closes. Calling this function makes the acquired systems available to other applications again. It is important to close initialized systems. Not closed systems will cause a *resource leak*. An attempt to open an unclosed MCS again will fail because the connection is still hold by the previous initialization.

`SA_CloseSystem` does not close systems that have been initialized with `SA_InitSystems`.

Parameters:

- *systemIndex* (SA_INDEX), input – A handle to the system which will be closed.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if(result == SA_OK){
	//Closing previously aquired system
	SA_CloseSystem(mcsHandle);
}
```

See also: `SA_OpenSystem`, `SA_GetSystemLocator`, `SA_FindSystems`

SA_ClearInitSystemsList

Caution: This function has been deprecated. Can only be used in conjunction with `SA_InitSystems`. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_ClearInitSystemsList();
```

Description:

This function may be used when acquiring specific systems at initialization. It clears the system initialization list. If `SA_InitSystems` is called after this, all available systems will be acquired.

Parameters:

None

Example:

```
SA_ClearInitSystemsList();
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);
if (result == SA_OK) {
	// All available systems were acquired.
	// Use SA_GetNumberOfSystems and SA_GetSystemID to see
	// how many and which systems these are.
}
```

See also: `SA_GetAvailableSystems`, `SA_AddSystemToInitSystemsList`, `SA_InitSystems`

SA_FindSystems

Interface:

```
SA_STATUS SA_FindSystems(const char *options,
                         char *outBuffer,
                         unsigned int *ioBufferSize);
```

Description:

This function writes a list of **locator** strings of MCS devices that are connected to the PC into *outBuffer*. Currently the function only lists MCS with a USB interface. O*ptions* contains a list of configuration options for the find procedure (currently unused). The caller must pass a pointer to a `char` buffer in *outBuffer* and set *ioBufferSize* to the size of the buffer. After the call the function has written a list of system locators into *outBuffer* and the number of written bytes into *ioBufferSize*. If the supplied buffer is too small to contain the generated list, the buffer will contain no valid content but *ioBufferSize* contains the required buffer size.

Parameters:

- *options* (const char), input – Options for the find procedure. **Currently unused.**
- *outBuffer* (char), output – Pointer to a buffer which holds the device locators after the function has returned
- *ioBufferSize* (unsigned int), input/output – Specifies the size of *outBuffer* before the function call. After the function call it holds the number of bytes written to *outBuffer*.

Example:

```
char outBuffer[4096];
unsigned int bufferSize = sizeof(outBuffer);
SA_STATUS result = SA_FindSystems("", outBuffer, &bufferSize);
if(result == SA_OK){
	// outBuffer holds the locator strings, separated by '\n'
	// bufferSize holds the number of bytes written to outBuffer
}
```

See also: Initialization, SA_OpenSystem

SA_GetAvailableSystems

Caution: This function has been deprecated. Please use the function `SA_FindSystems` for a list of available systems. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_GetAvailableSystems(unsigned int *idList,
                                 unsigned int *idListSize);
```

Description:

This is an informational function only. It has no side effects. It may be used to retrieve a list of systems that are connected to the PC and ready for acquisition at the time of the function call. This function is only callable if no systems have been acquired yet by `SA_InitSystems`.
The function receives two pointers. The first one must point to an array that will hold the resulting system IDs. The second one must point to a value that holds the size of the array so the function can assure not to overwrite unreserved memory space. The function looks for available systems and then checks whether the array is large enough to hold all resulting system IDs. If the array is too small the function will return an `SA_ID_LIST_TOO_SMALL_ERROR`. Otherwise it will write the IDs to the array and set the *idListSize* parameter to the number of elements written.

Parameters:

- *idList* (unsigned 32bit), output - This parameter must be a pointer to an array of 32bit values. If the function call was successful, the list holds the system IDs of all connected systems that are ready for acquisition.
- *idListSize* (unsigned 32bit), input/output - This parameter must hold the size of the array that the *idList* pointer points to. When the function returns, this parameter will always hold the number of system IDs that have been written to the array. If an error occurred, it will be zero.

Example:

```
#define LIST_SIZE 16
unsigned int idList[LIST_SIZE];
unsigned int listSize = LIST_SIZE;
SA_STATUS result = SA_GetAvailableSystems(idList,&listSize);
if (result == SA_OK) {
       // listSize holds the number of systems
       // idList array holds the systems IDs
}
```

See also: `SA_AddSystemToInitSystemsList`, `SA_ClearInitSystemsList`, `SA_InitSystems`

SA_GetChannelType

Interface:

```
SA_STATUS SA_GetChannelType(SA_INDEX systemIndex,
                            SA_INDEX channelIndex,
                            unsigned int *type);
```

Description:

This function may be used to determine the type of a channel of a system. Each channel of a system has a specific type. Currently there are two types of channels: positioner channels and end effector channels. Most functions of section II are only callable for certain channel types. The function descriptions list for which types they may be called.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *type* (unsigned 32bit), output - If the call was successful this parameter holds the channel type of the selected channel. Possible values are `SA_POSITIONER_CHANNEL_TYPE` and `SA_END_EFFECTOR_CHANNEL_TYPE`.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
unsigned int type;
result = SA_GetChannelType(mcsHandle,0,&type);
if (result == SA_OK) {
      // type holds the channel type of the first channel of the first system
}
```

SA_GetDLLVersion

Interface:

```
SA_STATUS SA_GetDLLVersion(unsigned int *version);
```

Description:

This function may be called to retrieve the version code of the library. It is useful to check if changes have been made to the software interface. An application may check the version in order to ensure that the library behaves as the application expects it to do.

The returned 32bit code is divided into three fields:

This function does not require the library to be initialized (see `SA_OpenSystem`) and will always return a status code of `SA_OK`.

Parameters:

- *version* (unsigned 32bit), output - Holds the version code. The higher the value the newer the version.

Example:

```
unsigned int version;
SA_GetDLLVersion(&version);
```

SA_GetInitState

Caution: This function has been deprecated. Can only be used in conjunction with `SA_InitSystems`. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_GetInitState(unsigned int *initMode);
```

Description:

This function may be used to check the initialization state of the library. If the library is not initialized then the returned mode will be `SA_INIT_STATE_NONE`. Otherwise either `SA_INIT_STATE_SYNC` or `SA_INIT_STATE_ASYNC`, depending on which parameter you passed to `SA_InitSystems`.

The return status of this function will always be `SA_OK`.

Parameters:

- *initMode* (unsigned 32bit), output - Returns the current initialization mode of the library. Will be one of `SA_INIT_STATE_NONE`, `SA_INIT_STATE_SYNC` or `SA_INIT_STATE_ASYNC`.

Example:

```
unsigned int mode;
SA_GetInitState(&mode);
if (mode == SA_INIT_STATE_NONE) {
	SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);
	if (result == SA_OK) {
		SA_GetInitState(&mode);
		// mode is SA_INIT_STATE_SYNC
	}
}
```

See also: `SA_InitSystems`, `SA_ReleaseSystems`

SA_GetNumberOfChannels

Interface:

```
SA_STATUS SA_GetNumberOfChannels(SA_INDEX systemIndex,
                                 unsigned int *channels);
```

Description:

This function may be used to determine how many channels are available on a system. This includes positioner channels and end effector channels. Each channel is of a specific type. Use the `SA_GetChannelType` function to determine the types of the channels.

Note that the number of channels does not represent the number positioners and/or end effectors that are currently connected to the system.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channels* (unsigned 32bit), output - If the call was successful this value holds the number of positioners and/or end effectors that may be connected to the given system.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
	// handle error...
}
unsigned int number;
result = SA_GetNumberOfChannels(mcsHandle,&number);
if (result == SA_OK) {
	// number holds the number of channels of the system
}
```

SA_GetNumberOfSystems

Caution: This function has been deprecated. Can only be used in conjunction with `SA_InitSystems`. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_GetNumberOfSystems(unsigned int *number);
```

Description:

This function may be used to determine how many systems have been acquired by a previously and successfully called `SA_InitSystems`.

Parameters:

- *number* (unsigned 32bit), output - If the call was successful this value holds the number of systems detected.

Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);
if (result != SA_OK) {
	// handle error...
}
unsigned int number;
result = SA_GetNumberOfSystems(&number);
if (result == SA_OK {
	// number holds the number of systems that were acquired
}
```

SA_GetSystemID

Caution: This function has been deprecated. Can only be used in conjunction with `SA_InitSystems`. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_GetSystemID(SA_INDEX systemIndex,
                         unsigned int *id);
```

Description:

This function may be used to physically identify a system connected to the PC. Each system has a unique ID which makes it possible to distinguish one from another. Once systems have been acquired by `SA_InitSystems` you may call this function to read out the system IDs of the acquired systems.

Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *id* (unsigned 32bit), output - If the call was successful this value holds the system ID of the selected system. The ID is a generic, unique number.

Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);
if (result != SA_OK) {
	// handle error...
}
unsigned int id;
result = SA_GetSystemID(0,&id);
if (result == SA_OK) {
	// id holds the system ID
}
```

SA_GetSystemLocator

Interface:

```
SA_STATUS SA_GetSystemLocator(SA_INDEX systemIndex,
                              char *outBuffer,
                              unsigned int *ioBufferSize);
```

Description:

Returns the locator of the initialized system *systemIndex* in *outBuffer.* When calling this function the caller must pass a buffer with a sufficient size and write the buffer size in *ioBufferSize*. After the call the function has written the system locator into *outBuffer* and the number of bytes written into *ioBufferSize*.

Parameters:

- *systemIndex* (SA_INDEX), input – Handle to an initialized system.
- *outBuffer* (char pointer), output – Pointer to a buffer which holds the device locator after the function has returned
- *ioBufferSize* (unsigned int pointer), input/output – Specifies the size of *outBuffer* before the function call. After the function call it holds the number of bytes written to *outBuffer*.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
char outBuffer[4096];
unsigned int bufferSize = sizeof(outBuffer);
SA_STATUS result = SA_GetSystemLocator(mcsHandle, outBuffer, &bufferSize);
if(result == SA_OK){
      // outBuffer holds the locator string
      // bufferSize holds the number of bytes written to outBuffer
}
```

See also: Initialization, `SA_OpenSystem`, `SA_FindSystems`

SA_InitSystems

Caution: This function has been deprecated. Please us the function `SA_OpenSystem` instead. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_InitSystems(unsigned int configuration);
```

Description:

This is the global initialization function. It must be called before any other functions are called. If the library is not initialized all other functions will return a `SA_NOT_INITIALIZED` status code. The only exceptions to this rule are the functions `SA_GetDLLVersion`, `SA_GetInitState`, `SA_GetAvailableSystems`, `SA_AddSystemToInitSystemsList` and `SA_ClearInitSystemsList`.

If the library is already initialized an implicit `SA_ReleaseSystems` call is made before (re-)initializing the library. If successful, all systems that are connected to the PC are acquired and can not be used by other software applications until `SA_ReleaseSystems` is called.

Optionally you may use the functions `SA_AddSystemToInitSystemsList` and `SA_ClearInitSystemsList` before calling `SA_InitSystems` to acquire specific systems.

When calling this function you must chose between one of two modes. If synchronous mode is selected, only functions of section IIa that have the suffix _S may be called thereafter. If asynchronous mode is selected, only functions of section IIb that have the suffix _A may be called.

Once systems have been acquired you may use the functions `SA_GetNumberOfSystems`, `SA_GetSystemID`, `SA_GetNumberOfChannels` and `SA_GetChannelType` (see there) to retrieve information about the systems that were acquired.

Parameters:

- *configuration* (unsigned 32bit), input - This parameter selects between the synchronous and the asynchronous communication. Possible values are `SA_SYNCHRONOUS_COMMUNICATION` and `SA_ASYNCHRONOUS_COMMUNICATION`. Optionally an `SA_HARDWARE_RESET` may be ORed to the value which sends a reset command to all systems. It has the same effect as a power-down/power-up cycle.

Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION | SA_HARDWARE_RESET);
if (result != SA_OK) {
	// handle error...
}
```

See also: `SA_ReleaseSystems`, `SA_GetInitState`

SA_OpenSystem

Interface:

```
SA_STATUS SA_OpenSystem(SA_INDEX *systemIndex,
                        const char *systemLocator,
                        const char *options);
```

Description:

Initializes one MCS specified in *systemLocator*, *systemIndex* is a handle to the opened system that is returned after a successful execution. It must be passed in the *systemIndex* parameter to the API functions. o*ptions* is a string parameter that contains a list of configuration options. The options must be separated by a comma or a newline.

The following options are available:

- *reset* — the MCS is reset on open. A reset has the same effect as a power-down/power-up cycle.
- *async*, *sync* — use the *async* option to set the communication mode to asynchronous, *sync* for synchronous communication. See “Communication Modes“ on p.8.
- *open-timeout <t>* — only available for network interfaces. <*t*> is the maximum time in milliseconds the PC tries to connect to the MCS. Default is 3000 milliseconds. The maximum timeout may be limited by operating system default parameters.

Systems that have been initialized with `SA_OpenSystem` must be released with `SA_CloseSystem`. `SA_ReleaseSystems` does **not** close them!

Parameters:

- *systemIndex* (SA_INDEX), output – returns a handle to the opened system.
- *systemLocator* (const char pointer), input – locator string that specifies the system.
- *options* (const char pointer), input – options for the initialization function. See list above.

Example:

```
const char loc1[] = "usb:id:3118167233";
const char loc2[] = "network:192.168.1.200:5000";


SA_STATUS result;
SA_INDEX mcsHandle1,mcsHandle2;

// connect to a USB interface for async. communication and reset the system
result = SA_OpenSystem(&mcsHandle1, loc1, "async,reset");
if(result != SA_OK){
	// handle error
}
// connect to a network interface for sync. communication with 1.5 sec timeout
result = SA_OpenSystem(&mcsHandle2, loc2, "sync,open-timeout 1500");
if(result != SA_OK){
	// handle error
}
```

See also: `SA_FindSystems`, `SA_GetSystemLocator`, `SA_CloseSystem`

SA_ReleaseSystems

Caution: This function has been deprecated. Can only be used in conjunction with `SA_InitSystems`. Please us the function `SA_CloseSystem` instead. This function may be removed from the API in the future!

Interface:

```
SA_STATUS SA_ReleaseSystems();
```

Description:

Inverse function to `SA_InitSystems`. This function should be called before the application closes. Calling this function makes the acquired systems available to other applications again.

Parameters:

None

Example:

```
SA_STATUS result = SA_ReleaseSystems();
```

See also: `SA_InitSystems`

SA_SetHCMEnabled

Interface:

```
SA_STATUS SA_SetHCMEnabled(SA_INDEX systemIndex,
                           unsigned int enabled);
```

Description:

This function may be used to enable or disable a Hand Control Module that is attached to the system to avoid interference while the software is in control of the system. (See also section ?? “Multiple Command Sources”.) There are three possible modes to set:

- `SA_HCM_DISABLED`: In this mode the Hand Control Module is disabled. It may not be used to control the system.
- `SA_HCM_ENABLED`: This is the default setting and the normal operation mode of the Hand Control Module.
- `SA_HCM_CONTROLS_DISABLED`: In this mode the Hand Control Module cannot be used to control the system. However, if there are positioners with sensors attached, their position data will still be displayed.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *enabled* (unsigned 32bit), input - Selects the mode. Must be one of `SA_HCM_DISABLED`, `SA_HCM_ENABLED` or `SA_HCM_CONTROLS_DISABLED`.

Example:

```
unsigned int mcsHandle;
const char loc[] = “usb:id:3118167233”;
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, “sync”);
if (result != SA_OK) {
       // handle error...
}
// disable the Hand Control Module
result = SA_SetHCMEnabled(mcsHandle,SA_HCM_DISABLED);
```

3.2 Functions for Synchronous Communication

General note: all functions of the synchronous communication mode are thread safe.

For functions that address a specific channel the channel type for which the function is callable is given.

SA_CalibrateSensor_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_CalibrateSensor_S(SA_INDEX systemIndex,
                               SA_INDEX channelIndex);
```

Description:

This function may be used to increase the accuracy of the position calculation. It 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error.

This function should 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 call this function on each initialization, but newly connected positioners have to be calibrated in order to ensure proper operation.

During the calibration the positioner will perform a movement of up to several mm. You must ensure, that the command is not executed while the positioner is near a mechanical end stop. Otherwise the calibration might fail and lead to unexpected behavior when executing closed-loop commands. As a safety precaution, also make sure that the positioner has enough freedom to move without damaging other equipment.

The calibration takes a few seconds to complete. During this time the positioner will report a status of `SA_CALIBRATING_STATUS` (see `SA_GetStatus_S`).

Positioners that are referenced via a mechanical end stop (see 5.4 “Sensor Types“) are moved to the end stop as part of the calibration routine. Which end stop is used for referencing is configured by `SA_SetSafeDirection_S`. Note that when changing the safe direction the end stop must be calibrated again for proper operation.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// start calibration routine
result = SA_CalibrateSensor_S(mcsHandle,0);
unsigned int status;
do {
      SA_GetStatus_S(mcsHandle,0,&status);
} while (status != SA_STOPPED_STATUS);
// done calibrating
```

SA_FindReferenceMark_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_FindReferenceMark_S(SA_INDEX systemIndex,
                                 SA_INDEX channelIndex,
                                 unsigned int direction,
                                 unsigned int holdTime
                                 unsigned int autoZero);
```

Description:

For positioners that are equipped with sensor feedback, this function may be used to move the positioner to a known physical position of the positioner. Some sensor types are equipped with a physical reference mark, others are referenced via a mechanical end stop (see appendix 5.4 “Sensor Types“). For latter types you must configure the safe direction with `SA_SetSafeDirection_S` and call `SA_CalibrateSensor_S` before the positioner can be properly referenced. The safe direction is then used instead of the direction parameter described below.

If the auto zero flag is set, the current position resp. angle is set to zero after the reference position has been reached. Otherwise the position is set according to the information stored in non-volatile memory of the last `SA_SetPosition_S` call. See section 2.5.3 “Defining Positions“ for more information.

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

The positioner may be instructed to hold the position of the reference mark after it has been reached. This behavior is similar to that of the other closed-loop commands, e.g. `SA_GotoPositionAbsolute_S`. See there for more information.

While executing the command the positioner will have a movement status of `SA_FINDING_REF_STATUS`. While holding the position the positioner will have a movement status of `SA_HOLDING_STATUS` (see `SA_GetStatus_S`).

If this command was successful, then the physical position of the positioner becomes known. See `SA_GetPhysicalPositionKnown_S`.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the system. The index is zero based.
- *direction* (unsigned 32bit), input - Specifies the initial search direction. Must be either `SA_BACKWARD_DIRECTION`, `SA_FORWARD_DIRECTION`, `SA_BACKWARD_FORWARD_DIRECTION` or `SA_FORWARD_BACKWARD_DIRECTION`. Note that this parameter is ignored for sensor types that are referenced via a mechanical end stop. Set the direction via `SA_SetSafeDirection_S` instead.
- *holdTime* (unsigned 32bit), input - Specifies how long (in milliseconds) the position is actively held after reaching the target. The valid range is 0..60,000. A 0 deactivates this feature, a value of 60,000 is infinite (until manually stopped, see `SA_Stop_S`).
- *autoZero* (unsigned 32bit), input - Must be one of `SA_NO_AUTO_ZERO` or `SA_AUTO_ZERO`. The latter will reset the current position resp. angle to zero upon reaching the reference mark (see also `SA_SetPosition_S`).

Example:

```
// move to reference mark and set to zero
result = SA_FindReferenceMark_S(mcsHandle,0,SA_FORWARD_DIRECTION,0,SA_AUTO_ZERO);
unsigned int status;
do {
       SA_GetStatus_S(mcsHandle,0,&status);
} while (status != SA_STOPPED_STATUS);
```

SA_GetAngle_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetAngle_S(SA_INDEX systemIndex,
                        SA_INDEX channelIndex,
                        unsigned int *angle,
                        signed int *revolution);
```

Description:

Returns the current angle of a positioner. This function 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error. Additionally, the command is only executable if the addressed channel is configured to be of type rotary (see `SA_SetSensorType_S`). A linear channel will return an error.

A rotary position is defined by a combination of an angle and a revolution. One revolution equals a full 360° turn. The angle value returned will always be in the range 0..359,999,999. If the positioner moves over a zero boundary, the angle value will wrap around accordingly and the revolution value will be incremented resp. decremented.

Please refer to section 2.5.1 “Rotary Sensors” for more information on the angle and revolution parameters.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *angle* (unsigned 32bit), output - If the call was successful this value holds the current angle of the positioner in micro degrees.
- *revolution* (signed 32bit), output - If the call was successful this value holds the current revolution of the positioner.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
	// handle error...
}
// get current position
signed int angle, revolution;
result = SA_GetAngle_S(mcsHandle,0,&angle,&revolution);
if (result == SA_OK) {
	// angle and revolution parameters have been updated
}
```

See also: `SA_GetPosition_S`

SA_GetAngleLimit_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetAngleLimit_S(SA_INDEX systemIndex,
                             SA_INDEX channelIndex,
                             unsigned int *minAngle,
                             signed int *minRevolution,
                             unsigned int *maxAngle,
                             signed int *maxRevolution);
```

Description:

Inverse function to `SA_SetAngleLimit_S`. May be used to read out the travel range limit that is currently configured for a rotary channel.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *minAngle* (unsigned 32bit), output - If the call was successful, this parameter holds the absolute minimum angle given in micro degrees.
- *minRevolution* (signed 32bit), output - If the call was successful, this parameter holds the absolute minimum revolution.
- *maxAngle* (unsigned 32bit), output - If the call was successful, this parameter holds the absolute maximum angle given in micro degrees.
- *maxRevolution* (signed 32bit), output - If the call was successful, this parameter holds the absolute maximum revolution.

Note: If no limit is set then all output parameters will be 0.

Example:

```
// retrieve the travel range limit of positioner 0
unsigned int min,max;
signed int minR, maxR;
result = SA_GetAngleLimit_S(mcsHandle,0,&min,&minR,&max,&maxR);
```

See also: `SA_SetAngleLimit_S`, `SA_GetPositionLimit_S`

SA_GetCaptureBuffer_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetCaptureBuffer_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                unsigned int bufferIndex,
                                SA_PACKET *buffer);
```

Description:

Retrieves the contents of a capture buffer. Capture buffers capture (groups of) internal values on certain events and may be used to synchronize the reading of values.

The function returns the data in form of an `SA_PACKET` (see header file). The *data1* field will always contain the buffer index that the packet represents (same as given in the *bufferIndex* parameter). The values of the other fields hold the capture buffer contents. See section 2.6.4 “Capture Buffers“ for a list of currently defined capture buffers.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *bufferIndex* (unsigned 32bit), input – Selects the buffer of the selected channel.
- *buffer* (`SA_PACKET`), output – If the call was successful this parameter holds the buffer contents.

Example:

```
SA_PACKET packet;
result = SA_GetCaptureBuffer_S(mcsHandle,0,0,&packet);
if (result == SA_OK) {
       // packet.data1 holds the buffer index (0)
       // packet.data2 holds the position value
       // packet.data3 holds the revolution value
       // packet.data4 holds the counter 0 value
}
```

SA_GetChannelProperty_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetChannelProperty_S(SA_INDEX systemIndex,
                                  SA_INDEX channelIndex,
                                  unsigned int key,
                                  signed int *value);
```

Description:

Retrieves a configuration value from a channel. This is a universal function to read out various channel properties. The property which is to be read is selected via the *key* parameter. This 32-bit parameter codes a combination of values and has the following structure:

31 24	23 16	15 8	7 0
component	sub component	property	

The `SA_EPK` helper function may be used to encode the key. See the header file for a list of component selectors and properties. The sub component selector is usually an index, but there can also be special sub component selectors. Note that not all properties are valid for all components. Please refer to section 2.4 “Channel Properties“ for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *key* (unsigned 32bit), input – Selects the property of which the value should be returned.
- *value* (signed 32bit), output – If the call was successful holds the value of the selected property.

Example:

```
// get current command queue size
unsigned int size;
result = SA_GetChannelProperty_S(
       mcsHandle,                              // system handle
       0,                                      // channel index
       SA_EPK(SA_COMMAND_QUEUE,0,SA_SIZE),     // property key
       &size                                   // property value
);
if (result == SA_OK) {
       // size holds the current command queue size
}
```

See also: `SA_SetChannelProperty_S`

SA_GetClosedLoopMoveAcceleration_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetClosedLoopMoveAcceleration_S(SA_INDEX systemIndex,
                                             SA_INDEX channelIndex,
                                             unsigned int *acceleration);
```

Description:

Returns the movement acceleration that is currently configured for a channel. See `SA_SetClosedLoopMoveAcceleration_S` for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *acceleration* (unsigned 32bit), output – If the call was successful, this parameter holds the movement acceleration that is currently configured for the given channel. The value is given in µm/s² for linear positioners and in m°/s² for rotary positioners. The valid range is 0..10,000,000. A value of 0 (default) indicates that the acceleration control feature is deactivated.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
// get current movement acceleration for channel 1 of system 0
unsigned int acceleration;
result = SA_GetClosedLoopMoveAcceleration_S(mcsHandle,1,&acceleration);
if (result == SA_OK) {
       // acceleration holds the current movement acceleration
}
```

See also: `SA_SetClosedLoopMoveAcceleration_S`, `SA_GetClosedLoopMoveSpeed_S`

SA_GetClosedLoopMoveSpeed_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetClosedLoopMoveSpeed_S(SA_INDEX systemIndex,
                                      SA_INDEX channelIndex,
                                      unsigned int *speed);
```

Description:

Returns the movement speed that is currently configured for a channel. See `SA_SetClosedLoopMoveSpeed_S` for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *speed* (unsigned 32bit), output - If the call was successful, this parameter holds the movement speed that is currently configured for the given channel. The value is given in nanometers per second for linear positioners and in micro degrees per second for rotary positioners. The valid range is 0..100,000,000. A value of 0 (default) means that the speed control feature is deactivated.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
// get current movement speed for channel 2
unsigned int speed;
result = SA_GetClosedLoopMoveSpeed_S(mcsHandle,2,&speed);
if (result == SA_OK) {
       // speed holds the current movement speed
}
```

See also: `SA_SetClosedLoopMoveSpeed_S`, `SA_GetClosedLoopMoveAcceleration_S`

SA_GetEndEffectorType_S

Channel type: End effector

Interface:

```
SA_STATUS SA_GetEndEffectorType_S(SA_INDEX systemIndex,
                                  SA_INDEX channelIndex,
                                  unsigned int *type,
                                  signed int *param1,
                                  signed int *param2);
```

Description:

This function may be used to retrieve the current end effector type configuration of an end effector channel.

See `SA_SetEndEffectorType_S` for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *type* (unsigned 32bit), output - If the call was successful this parameter holds the type of end effector that is currently configured for the channel. Possible values are `SA_GRIPPER_END_EFFECTOR_TYPE`, `SA_FORCE_SENSOR_END_EFFECTOR_TYPE` and `SA_FORCE_GRIPPER_END_EFFECTOR_TYPE`.
- *param1* (signed 32bit), output - The meaning and valid range of this parameter depends on the type (see table on page 79).
- *param2* (signed 32bit), output - The meaning and valid range of this parameter depends on the type (see table on page 79).

Example:

```
// get end effector type for channel three
unsigned int endEffectorType;
signed int param1, param2;
result = SA_GetEndEffectorType_S(mcsHandle,3,&endEffectorType,&param1,&param2);
if (result == SA_OK) {
      // refer to table on page 79 for the meaning of the output values
}
```

See also: `SA_SetEndEffectorType_S`

SA_GetForce_S

Channel type: End effector

Interface:

```
SA_STATUS SA_GetForce_S(SA_INDEX systemIndex,
                        SA_INDEX channelIndex,
                        signed int *force);
```

Description:

This command is only executable if the end effector that is connected to the select channel has a force sensor (see `SA_SetEndEffectorType_S`). The function returns the force that is currently measured by the sensor. Note that the sensor must be calibrated in order to provide proper measurement values. See `SA_SetZeroForce_S`.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *force* (signed 32bit), output - If the call was successful this value holds the force that is currently measured by the sensor. The value is given in 1/10 µN.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       // handle error...
}
// get current status
unsigned int status;
result = SA_GetForce_S(mcsHandle,0,&status);
```

See also: `SA_SetZeroForce_S`

SA_GetGripperOpening_S

Channel type: End effector

Interface:

```
SA_STATUS SA_GetGripperOpening_S(SA_INDEX systemIndex,
                                 SA_INDEX channelIndex,
                                 unsigned int *opening);
```

Description:

This command is only executable if the end effector that is connected to the selected channel is a gripper (see `SA_SetEndEffectorType_S`). The function returns the voltage that is currently applied to the gripper.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *opening* (unsigned 32bit), output - If the call was successful this value holds the current voltage that is applied to the gripper. The value is given in 1/100 Volts.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       // handle error...
}
// get current voltage
unsigned int voltage;
result = SA_GetGripperOpening_S(mcsHandle,0,&voltage);
```

See also: `SA_GotoGripperOpeningAbsolute_S`, `SA_GotoGripperOpeningRelative_S`

SA_GetPhysicalPositionKnown_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetPhysicalPositionKnown_S(SA_INDEX systemIndex,
                                        SA_INDEX channelIndex,
                                        unsigned int *known);
```

Description:

Returns whether the positioner “knows” its physical position. After a power-up the physical position is unknown and the current position is implicitly assumed to be the zero position. After the reference mark has been found by calling `SA_FindReferenceMark_S` the physical position becomes known.

This function can be useful if the software application restarts and connects to a system that has stayed online. If the physical position is already known, traveling to the reference mark again may be omitted.

See also 2.5.3 “Defining Positions“.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *known* (unsigned 32bit), output - If the call was successful, this parameter will be either `SA_PHYSICAL_POSITION_UNKNOWN` or `SA_PHYSICAL_POSITION_KNOWN`.

Example:

```
// check whether the physical position of channel 2 is known
unsigned int known;
result = SA_GetPhysicalPositionKnown_S(mcsHandle,2,&known);
if (result == SA_OK) {
      // known holds the result
}
```

See also: `SA_FindReferenceMark_S`, `SA_SetPosition_S`, `SA_SetScale_S`

SA_GetPosition_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetPosition_S(SA_INDEX systemIndex,
                           SA_INDEX channelIndex,
                           signed int *position);
```

Description:

Returns the current position of a positioner. This function 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error. Additionally, the command is only executable if the addressed channel is configured to be of type linear (see `SA_SetSensorType_S`). A rotary channel will return an error (use `SA_GetAngle_S` instead).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *position* (signed 32bit), output - If the call was successful this value holds the current position of the positioner in nano meters.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
	// handle error...
}
// get current position
signed int position;
result = SA_GetPosition_S(mcsHandle,0,&position);
if (result == SA_OK) {
	// current position is in position variable
}
```

See also: `SA_GetAngle_S`

SA_GetPositionLimit_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetPositionLimit_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                signed int *minPosition,
                                signed int *maxPosition);
```

Description:

Inverse function to `SA_SetPositionLimit_S`. May be used to read out the travel range limit that is currently configured for a linear channel.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *minPosition* (signed 32bit), output - If the call was successful, this parameter holds the absolute minimum position given in nanometers.
- *maxPosition* (signed 32bit), output - If the call was successful, this parameter holds the absolute maximum position given in nanometers.

Note: If no limit is set then both *minPosition* and *maxPosition* will be 0.

Example:

```
// retrieve the travel range limit of positioner 0
signed int min,max;
result = SA_GetPositionLimit_S(mcsHandle,0,&min,&max);
if (result == SA_OK) {
       // min and max hold the range limit
}
```

See also: `SA_SetPositionLimit_S`, `SA_GetAngleLimit_S`

SA_GetSafeDirection_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetSafeDirection_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                unsigned int *direction);
```

Description:

Retrieves the currently configured safe direction of a channel. See `SA_SetSafeDirection_S` for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *direction* (unsigned 32bit), output - If the call was successful, this parameter holds the currently configured safe direction. Will be either `SA_FORWARD_DIRECTION` or `SA_BACKWARD_DIRECTION`.

Example:

```
unsigned int direction;
SA_STATUS result = SA_GetSafeDirection_S(mcsHandle,1,&direction);
if (result == SA_OK) {
	// direction holds the safe direction
}
```

See also: `SA_SetSafeDirection_S`

SA_GetScale_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetScale_S(SA_INDEX systemIndex,
                        SA_INDEX channelIndex,
                        signed int *scale,
                        unsigned int *inverted);
```

Description:

Retrieves the currently configured scale of a channel. See `SA_SetScale_S` for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *scale* (signed 32bit), output - If the call was successful, this parameter holds the currently configured scale.
- *inverted* (unsigned 32bit) – If the call was successful, this parameter hold the currently configured scale inversion. Will be either `SA_FALSE` or `SA_TRUE`.

Caution: Please note that only the logical scale of the positioner will be inverted when the `inverted` value has changed. Parameters like the *SafeDirection* will not be altered. Thus the positioner will move in the opposite direction when e.g. calling SA_FindReferenceMark_S with the same parameters prior to the inversion change.

Example:

```
signed int scale;
unsigned int inverted;
SA_STATUS result = SA_GetScale_S(mcsHandle,1,&scale,&inverted);
if (result == SA_OK) {
       // scale holds the configured scale
       // inverted holds the scale inversion setting
}
```

See also: `SA_SetScale_S`

SA_GetSensorEnabled_S

Interface:

```
SA_STATUS SA_GetSensorEnabled_S(SA_INDEX systemIndex,
                                unsigned int *enabled);
```

Description:

This function may be used to read the sensor operation mode that is currently configured for the sensors that are attached to the positioners of a system. The mode is system global and applies to all positioner channels of a system equally.

Please refer to section 2.5.2 “Sensor Modes” for more information on the sensor modes.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *enabled* (unsigned 32bit), output - If the call was successful, this parameter holds the current sensor mode (`SA_SENSOR_DISABLED`, `SA_SENSOR_ENABLED` or `SA_SENSOR_POWERSAVE`).

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       // handle error...
}
// read sensor mode
unsigned int enabled;
result = SA_GetSensorEnabled_S(mcsHandle,&enabled);
```

See also: `SA_SetSensorEnabled_S`

SA_GetSensorType_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetSensorType_S(SA_INDEX systemIndex,
                             SA_INDEX channelIndex,
                             unsigned int *type);
```

Description:

Returns the type of sensor that is configured for the given channel (see `SA_SetSensorType_S`). The returned type will be one of the types listed in the appendix (5.4 “Sensor Types“).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *type* (unsigned 32bit), output - If the call was successful, this parameter holds the type of the sensor.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       // handle error...
}
// get sensor type for second channel
unsigned int sensorType;
result = SA_GetSensorType_S(mcsHandle,1,&sensorType);
```

See also: `SA_SetSensorType_S`, `SA_GetPosition_S`, `SA_GetAngle_S`

SA_GetStatus_S

Channel type: Positioner, End effector

Interface:

```
SA_STATUS SA_GetStatus_S(SA_INDEX systemIndex,
                         SA_INDEX channelIndex,
                         unsigned int *status);
```

Description:

Returns the current movement status of a positioner or end effector (see appendix 5.3 “Channel Status Codes“ for a list of movement status codes). This function can be used to check whether a previously issued movement command has been completed.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *status* (unsigned 32bit), output - If the call was successful this value holds the current status of the positioner. Please refer to the appendix for a list of status codes.

Example:

```
unsigned int mcsHandle;
const char loc[] = “usb:id:3118167233”;
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, “sync”);
if (result != SA_OK) {
      // handle error...
}
// get current status
unsigned int status;
result = SA_GetStatus_S(0,0,&status);
```

SA_GetVoltageLevel_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetVoltageLevel_S(SA_INDEX systemIndex,
                               SA_INDEX channelIndex,
                               unsigned int *level);
```

Description:

Returns the voltage level that is currently applied to the piezo element of a positioner. This function is mainly of interest in conjunction with `SA_ScanMoveAbsolute_S` and `SA_ScanMoveRelative_S`, since these are used to control the voltage level.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *level* (unsigned 32bit), output - If the call was successful this value holds the current voltage level that is applied to the piezo element of the positioner. It ranges from 0..4,095, where 0 corresponds to 0V and 4,095 to 100V.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// get current voltage level
unsigned int level;
result = SA_GetVoltageLevel_S(mcsHandle,0,&level);
if (result == SA_OK) {
      // voltage level is in level variable
}
```

See also: `SA_ScanMoveAbsolute_S`, `SA_ScanMoveRelative_S`

SA_GotoAngleAbsolute_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GotoAngleAbsolute_S(SA_INDEX systemIndex,
                                 SA_INDEX channelIndex,
                                 unsigned int angle,
                                 signed int revolution,
                                 unsigned int holdTime);
```

Description:

Instructs a positioner to turn to a specific angle. This function 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 `SA_SetSensorEnabled_S`). If this is not the case an error will be returned. Additionally, the command is only executable if the addressed channel is configured to be of type rotary (see `SA_SetSensorType_S`). A linear channel will return an error (use `SA_GotoPositionAbsolute_S` instead).

The positioner may be instructed to hold the target angle after it has been reached. This may be useful to compensate for drift effects and the like. Note that the positioner will use the scan mode to hold the angle if needed. When the piezo element of the positioner reaches a scanning boundary a single step is performed. However, if this behavior is not desired the correction steps can be disabled with the `SA_SetStepWhileScan_S` command (see there). Note though that disabling the steps might mean that the angle cannot be held.

While executing the command the positioner will have a movement status of `SA_TARGET_STATUS`. While holding the target position the positioner will have a movement status of `SA_HOLDING_STATUS` (see `SA_GetStatus_S`).

If a mechanical end stop is detected while the command is in execution, the movement will be aborted. Note that in asynchronous communication mode an error will be reported.

Please refer to section 2.5.1 “Rotary Sensors” for more information on the angle and revolution parameters.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *angle* (unsigned 32bit), input - Absolute angle to move to in micro degrees. The valid range is 0..359,999,999.
- *revolution* (signed 32bit), input - Absolute revolution to move to. The valid range is -32,768..32,767.
- *holdTime* (unsigned 32bit), input - Specifies how long (in milliseconds) the angle is actively held after reaching the target. The valid range is 0..60,000. A 0 deactivates this feature, a value of 60,000 is infinite (until manually stopped, see `SA_Stop_S`).

Example:

```
// move to 90° angle and hold position for one second
result = SA_GotoAngleAbsolute_S(mcsHandle,0,90000000,0,1000);
```

See also: `SA_GotoAngleRelative_S`, `SA_GotoPositionAbsolute_S`, `SA_GetAngle_S`

SA_GotoAngleRelative_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GotoAngleRelative_S(SA_INDEX systemIndex,
                                 SA_INDEX channelIndex,
                                 signed int angleDiff,
                                 signed int revolutionDiff,
                                 unsigned int holdTime);
```

Description:

Instructs a positioner to move to an angle relative to its current angle. This function 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error. Additionally, the command is only executable if the addressed channel is configured to be of type rotary (see `SA_SetSensorType_S`). A linear channel will return an error (use `SA_GotoPositionRelative_S` instead).

If a relative positioning command is issued before a previous one has finished, normally the relative targets are accumulated. If this is not desired it can be disabled with `SA_SetAccumulateRelativePositions_S` (see there for more information).

The positioner may be instructed to hold the target angle after it has been reached. See `SA_GotoAngleAbsolute_S` for more information.

While executing the command the positioner will have a movement status of `SA_TARGET_STATUS`. While holding the target angle the positioner will have a movement status of `SA_HOLDING_STATUS` (see `SA_GetStatus_S`).

If a mechanical end stop is detected while the command is in execution, the movement will be aborted. Note that in asynchronous communication mode an error will be reported.

Please refer to section 2.5.1 “Rotary Sensors” for more information on the angle and revolution parameters.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *angleDiff* (signed 32bit), input - Relative angle to move to in micro degrees. The valid range is -359,999,999..359,999,999.
- *revolutionDiff* (signed 32bit), input - Relative revolution to move to. The valid range is -32,768..32,767.
- *holdTime* (unsigned 32bit), input - Specifies how long (in milliseconds) the angle is actively held after reaching the target. The valid range is 0..60,000. A 0 deactivates this feature, a value of 60,000 is infinite (until manually stopped, see `SA_Stop_S`).

Example:

```
// do one full turn plus another 45° in negative direction
result = SA_GotoAngleRelative_S(mcsHandle,0,-45000000,-1,0);
```

See also: `SA_GotoAngleAbsolute_S`, `SA_GotoPositionRelative_S`, `SA_GetAngle_S`

SA_GotoGripperForceAbsolute_S

Channel type: End effector

Interface:

```
SA_STATUS SA_GotoGripperForceAbsolute_S(SA_INDEX systemIndex,
                                        SA_INDEX channelIndex,
                                        signed int force,
                                        unsigned int speed,
                                        unsigned int holdTime);
```

Description:

This closed-loop command is only executable if the end effector that is connected to the channel is a gripper with an integrated force sensor. The function may be used to grab an object with a defined and constant force. The channel will adjust the output voltage of the gripper so that the force sensor measures the given force.

Note that the force sensor must be calibrated in order for this command to function properly. See `SA_SetZeroForce_S`.

While executing the command the end effector will have a status of `SA_TARGET_STATUS` (see `SA_GetStatus_S`). Once the target force is reached it will try to hold the given force and have a status of `SA_HOLDING_STATUS` until the channel is explicitly stopped (see `SA_Stop_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *force* (signed 32bit), input - Specifies the force that is to be applied. It is given in tenths of µN, e.g. a value of 100 would be 10µN. The valid range for this parameter is -100,000 ..100,000.
- *speed* (unsigned 32bit), input - This parameter may be used to limit the speed with which the gripper is opened or closed. It is given in Volts per second. The valid range for this parameter is 1 .. 225,000.
- *holdTime* (unsigned 32bit), input - THIS PARAMETER HAS NOT BEEN IMPLEMENTED YET.

Example:

```
// grab with 20µN, do not move the gripper faster than 100 V/s
SA_GotoGripperForceAbsolute_S(mcsHandle,0,200,100,0);
```

See also: `SA_SetZeroForce_S`

SA_GotoGripperOpeningAbsolute_S

Channel type: End effector

Interface:

```
SA_STATUS SA_GotoGripperOpeningAbsolute_S(SA_INDEX systemIndex,
                                          SA_INDEX channelIndex,
                                          unsigned int opening,
                                          unsigned int speed);
```

Description:

For end effectors that have a gripper this function may be used to open or close the gripper. For this a voltage is applied to it. Applying 0V will open the gripper all the way. The higher the applied voltage the further the gripper will close. The maximum allowed voltage depends on the gripper type (see `SA_SetEndEffectorType_S`). If the given voltage is higher than the allowed voltage for the currently configured end effector the channel will stop at the maximum voltage.

While executing the command the end effector will have a status of `SA_OPENING_STATUS` (see `SA_GetStatus_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *opening* (unsigned 32bit), input - Specifies the target voltage which is given in 1/100 Volts, e.g. a value of 10,000 would be 100V. The valid range for this parameter is 0 .. 22,500.
- *speed* (unsigned 32bit), input - Specifies the speed of the voltage adjustment. It is given in Volts per second. The valid range for this parameter is 1.. 225,000.

Example:

```
unsigned int status;
// open gripper all the way (very fast)
SA_GotoGripperOpeningAbsolute_S(mcsHandle,0,0,10000);
// wait until gripper is open
do {
       SA_GetStatus_S(mcsHandle,0,&status);
} while (status != SA_STOPPED_STATUS);
// close gripper to 50V within two seconds
SA_GotoGripperOpeningAbsolute_S(mcsHandle,0,5000,25);
```

See also: `SA_GotoGripperOpeningRelative_S`

SA_GotoGripperOpeningRelative_S

Channel type: End effector

Interface:

```
SA_STATUS SA_GotoGripperOpeningRelative_S(SA_INDEX systemIndex,
                                          SA_INDEX channelIndex,
                                          signed int diff,
                                          unsigned int speed);
```

Description:

This function is similar to `SA_GotoGripperOpeningAbsolute_S` (see there) with the difference that a relative movement is performed. If the resulting target voltage exceeds the allowed range (below zero or above the maximum voltage for the currently configured gripper type) the channel will stop at the boundary.

While executing the command the end effector will have a status of `SA_OPENING_STATUS` (see `SA_GetStatus_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *diff* (signed 32bit), input - Specifies the relative target voltage which is given in 1/100 Volts, e.g. a value of -1,000 would "open the gripper by 10V". The valid range for this parameter is -22,500 .. 22,500.
- *speed* (unsigned 32bit), input - Specifies the speed of the voltage adjustment. It is given in Volts per second. The valid range for this parameter is 1.. 225,000.

Example:

```
unsigned int status;
// open gripper all the way (very fast)
SA_GotoGripperOpeningAbsolute_S(mcsHandle,0,0,10000);
// wait until gripper is open
do {
       SA_GetStatus_S(mcsHandle,0,&status);
} while (status != SA_STOPPED_STATUS);
// close gripper by 25V within half a second
SA_GotoGripperOpeningRelative_S(mcsHandle,0,2500,50);
```

See also: `SA_GotoGripperOpeningAbsolute_S`

SA_GotoPositionAbsolute_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GotoPositionAbsolute_S(SA_INDEX systemIndex,
                                    SA_INDEX channelIndex,
                                    signed int position,
                                    unsigned int holdTime);
```

Description:

Instructs a positioner to move to a specific position. This function 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 `SA_SetSensorEnabled_S`). If this is not the case an error will be returned. Additionally, the command is only executable if the addressed channel is configured to be of type linear (see `SA_SetSensorType_S`). A rotary channel will return an error (use `SA_GotoAngleAbsolute_S` instead).

The positioner 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. Note that the positioner will use the scan mode to hold the position if needed. When the piezo element of the positioner reaches a scanning boundary a single step is performed. However, if this behavior is not desired the correction steps can be disabled with the `SA_SetStepWhileScan_S` command (see there). Note though that disabling the steps might mean that the position cannot be held.

While executing the command the positioner will have a movement status of `SA_TARGET_STATUS`. While holding the target position the positioner will have a movement status of `SA_HOLDING_STATUS` (see `SA_GetStatus_S`).

If a mechanical end stop is detected while the command is in execution, the movement will be aborted. Note that in asynchronous communication mode an error will be reported.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *position* (signed 32bit), input - Absolute position to move to in nano meters.
- *holdTime* (unsigned 32bit), input - Specifies how long (in milliseconds) the position is actively held after reaching the target. The valid range is 0..60,000. A 0 deactivates this feature, a value of 60,000 is infinite (until manually stopped, see `SA_Stop_S`).

Example:

```
// move to zero position
result = SA_GotoPositionAbsolute_S(mcsHandle,0,0,0);
```

See also: `SA_GotoPositionRelative_S`, `SA_GotoAngleAbsolute_S`, `SA_GetPosition_S`

SA_GotoPositionRelative_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_GotoPositionRelative_S(SA_INDEX systemIndex,
                                    SA_INDEX channelIndex,
                                    signed int diff,
                                    unsigned int holdTime);
```

Description:

Instructs a positioner to move to a position relative to its current position. This function 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error. Additionally, the command is only executable if the addressed channel is configured to be of type linear (see `SA_SetSensorType_S`). A rotary channel will return an error (use `SA_GotoAngleRelative_S` instead).

If a relative positioning command is issued before a previous one has finished, normally the relative targets are accumulated. If this is not desired it can be disabled with `SA_SetAccumulateRelativePositions_S` (see there for more information).

The positioner may be instructed to hold the target position after it has been reached. See `SA_GotoPositionAbsolute_S` for more information.

While executing the command the positioner will have a movement status of `SA_TARGET_STATUS`. While holding the target position the positioner will have a movement status of `SA_HOLDING_STATUS` (see `SA_GetStatus_S`).

If a mechanical end stop is detected while the command is in execution, the movement will be aborted. Note that in asynchronous communication mode an error will be reported.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *position* (signed 32bit), input - Relative position to move to in nano meters.
- *holdTime* (unsigned 32bit), input - Specifies how long (in milliseconds) the position is actively held after reaching the target. The valid range is 0..60,000. A 0 deactivates this feature, a value of 60,000 is infinite (until manually stopped, see `SA_Stop_S`).

Example:

```
// move 2 micro meters in negative direction
result = SA_GotoPositionRelative_S(mcsHandle,0,-2000,0);
```

See also: `SA_GotoPositionAbsolute_S`, `SA_GotoAngleRelative_S`, `SA_GetPosition_S`

SA_ScanMoveAbsolute_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_ScanMoveAbsolute_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                unsigned int target,
                                unsigned int scanSpeed);
```

Description:

Performs a scanning movement of a positioner to a specific target scan position. This function may be used to directly control the deflection of the piezo of the positioner.

While executing the command the positioner will have a movement status of `SA_SCANNING_STATUS` (see `SA_GetStatus_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *target* (unsigned 32bit), input - Target scan position to which to scan to. The value is given as a 12bit value (range 0..4,095). 0 corresponds to 0V, 4,095 to 100V.
- *scanSpeed* (unsigned 32bit), input - The valid range is 1 .. 4,095,000,000 and represents single 12bit increments per second. With a value of 1 a scan over the full range from 0 to 4,095 takes 4,095 seconds while at full speed the scan is performed in one micro second.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// scan to 50V
result = SA_ScanMoveAbsolute_S(mcsHandle,0,2048,10000);
// scan to 0V within two seconds
result = SA_ScanMoveAbsolute_S(mcsHandle,0,0,1024);
```

See also: `SA_ScanMoveRelative_S`, `SA_GetVoltageLevel_S`

SA_ScanMoveRelative_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_ScanMoveRelative_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                signed int diff,
                                unsigned int scanSpeed);
```

Description:

Performs a relative scanning movement of a positioner.

While executing the command the positioner will have a movement status of `SA_SCANNING_STATUS` (see `SA_GetStatus_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *diff* (signed 32bit), input - Relative scan target to which to scan to. The valid range is -4,095 .. 4,095. If the resulting absolute scan target exceeds the valid range of 0..4,095 the scan movement will stop at the boundary.
- *scanSpeed* (unsigned 32bit), input - The valid range is 1 .. 4,095,000,000 and represents single 12bit increments per second. With a value of 1 a scan over the full range from 0 to 4,095 takes 4,095 seconds while at full speed the scan is performed in one micro second.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// scan to 50V
result = SA_ScanMoveAbsolute_S(mcsHandle,0,2048,10000);
// scan to 25V within one second
result = SA_ScanMoveRelative_S(mcsHandle,0,-1024,1024);
// scan to (automatically stop at) 0V
result = SA_ScanMoveRelative_S(mcsHandle,0,-3000,1024);
```

See also: `SA_ScanMoveAbsolute_S`, `SA_GetVoltageLevel_S`

SA_SetAccumulateRelativePositions_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetAccumulateRelativePositions_S(SA_INDEX systemIndex,
                                              SA_INDEX channelIndex,
                                              unsigned int accumulate);
```

Description:

This function is of interest in conjunction with the closed-loop commands `SA_GotoPositionRelative_S` and `SA_GotoAngleRelative_S` (see there). It sets a flag that 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 all new relative position commands are added to the previous target position. Otherwise 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 +1mm as relative target. With accumulation active the final position will be 2mm. With accumulation inactive the final position will vary (e.g. 1.12mm) depending on when the second command arrives.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *accumulate* (unsigned 32bit), input - Selects the mode. Must be either `SA_NO_ACCUMULATE_RELATIVE_POSITIONS` or `SA_ACCUMULATE_RELATIVE_POSITIONS`. The latter is the default.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// disable accumulation of relative movement commands in closed-loop mode
result = SA_SetAccumulateRelativePositions_S(mcsHandle,
                                             0,SA_NO_ACCUMULATE_RELATIVE_POSITIONS);
```

See also: `SA_GotoPositionRelative_S`, `SA_GotoAngleRelative_S`

SA_SetAngleLimit_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetAngleLimit_S(SA_INDEX systemIndex,
                             SA_INDEX channelIndex,
                             unsigned int minAngle,
                             signed int minRevolution,
                             unsigned int maxAngle,
                             signed int maxRevolution);
```

Description:

For positioners with integrated sensors this function may be used to limit the travel range of a rotary positioner by software. (For linear positioners see `SA_SetPositionLimit_S`.) By default there is no limit set. If defined the positioner will not move beyond the limit. This affects open-loop as well as closed-loop movements.
Note that the limit may only be set if the physical position is known at the time of the call (see `SA_FindReferenceMark_S`, `SA_GetPhysicalPositionKnown_S`).

See section 2.5.1 “Rotary Sensors“ and 2.5.4 “Software Range Limit“ for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *minAngle* (unsigned 32bit), input - Absolute minimum angle given in micro degrees. The valid range is 0 .. 359,999,999.
- *minRevolution* (signed 32bit), input - Absolute minimum revolution. The valid range is -32768 .. 32767.
- *maxAngle* (unsigned 32bit), input - Absolute maximum angle given in micro degrees. The valid range is 0 .. 359,999,999.
- *maxRevolution* (signed 32bit), input - Absolute maximum revolution. The valid range is -32768 .. 32767.

Note: The *maxAngle* / *maxRevolution* pair must be greater than the *minAngle* / *minRevolution* pair, otherwise the positioner will not move at all. If both pairs have the same value then the software range limit is disabled.

Example:

```
// limit the travel range of positioner 0 to +/- 10° around the zero angle
result = SA_SetAngleLimit_S(mcsHandle,0,350000000,-1,10000000,0);
```

See also: `SA_GetAngleLimit_S`, `SA_SetPositionLimit_S`, `SA_FindReferenceMark_S`

SA_SetChannelProperty_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetChannelProperty_S(SA_INDEX systemIndex,
                                  SA_INDEX channelIndex,
                                  unsigned int key,
                                  signed int value);
```

Description:

Sets a configuration value of a channel. This is a universal function to configure various channel properties. The property which is to be set is selected via the *key* parameter. This 32-bit parameter codes a combination of values and has the following structure:

31 24	23 16	15 8	7 0
component	sub component	property	

The `SA_EPK` helper function may be used to encode the key. See the header file for a list of component selectors and properties. The sub component selector is usually an index, but there can also be special sub component selectors. Note that not all properties are valid for all components. Please refer to section 2.4 “Channel Properties“ for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *key* (unsigned 32bit), input – Selects the property of which the value should be set.
- *value* (signed 32bit), input – Defines the value that the selected property should have.

Example:

```
SA_STATUS result;
// reset event counter 0 to zero
result = SA_SetChannelProperty_S(
       mcsHandle,                       // system handle
       0,                               // channel index
       SA_EPK(SA_COUNTER,0,SA_VALUE),   // property key
       0                                // property value
);
```

See also: `SA_GetChannelProperty_S`

SA_SetClosedLoopMaxFrequency_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetClosedLoopMaxFrequency_S(SA_INDEX systemIndex,
                                         SA_INDEX channelIndex,
                                         unsigned int frequency);
```

Description:

For positioners that have a sensor installed, this function may be used to define the maximum frequency that the positioners are driven with when issuing closed-loop movement commands (e.g. `SA_GotoPositionAbsolute_S`). This parameter may be set for each channel independently. Once set, all subsequent closed-loop commands will execute with the new setting.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *frequency* (unsigned 32bit), input - Defines the maximum driving frequency in Hz. The valid range is 50 .. 18,500.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// set maximum closed-loop frequency of first positioner connected to the
// system to 3kHz.
result = SA_SetClosedLoopMaxFrequency_S(mcsHandle,0,3000);
```

SA_SetClosedLoopMoveAcceleration_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetClosedLoopMoveAcceleration_S(SA_INDEX systemIndex,
                                             SA_INDEX channelIndex,
                                             unsigned int acceleration);
```

Description:

This function configures the acceleration control feature of a channel for closed-loop commands such as `SA_GotoPositionAbsolute_S`. By default the acceleration control is inactive. If activated, all following closed-loop commands will be executed with the new acceleration.

Note that the acceleration control feature requires the speed control feature. Enabling acceleration control will implicitly enable the speed control should it be inactive.

Likewise, the acceleration control is required by the Low Vibration mode (see 2.4.2 “Low Vibration“). Disabling the acceleration control will cause the Low Vibration mode to be implicitly disabled as well.

See also section 2.7.2 “Dependency Chains“.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *acceleration* (unsigned 32bit), input - Sets the movement acceleration for closed-loop commands which is given in µm/s² for linear positioners and in m°/s² for rotary positioners. The valid range is 0..10,000,000. A value of 0 (default) deactivates the acceleration control feature.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
SA_STATUS result;
// set the acceleration to 1 mm/s²
result = SA_SetClosedLoopMoveAcceleration_S(mcsHandle,0,1000);
```

See also: `SA_GetClosedLoopMoveAcceleration_S`, `SA_SetClosedLoopMoveSpeed_S`

SA_SetClosedLoopMoveSpeed_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetClosedLoopMoveSpeed_S(SA_INDEX systemIndex,
                                      SA_INDEX channelIndex,
                                      unsigned int speed);
```

Description:

This function configures the speed control feature of a channel for closed-loop commands such as `SA_GotoPositionAbsolute_S`. By default the speed control is inactive. In this state the behavior of closed-loop commands is influenced by the maximum driving frequency (see `SA_SetClosedLoopMaxFrequency_S`). If a movement speed is configured, all following closed-loop commands will be executed with the new speed. Note that 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 movement speed, then the movement speed might not be reached or held. In this case increase the maximum frequency.

Be aware that different positioners reach different speeds. If a positioner is not able to move as fast as the configured move speed, then the driver will cap at the maximum driving frequency.

Note that the speed control feature is required by the acceleration control feature (see `SA_SetClosedLoopMoveAcceleration_S`). Disabling the speed control will cause the acceleration control to be implicitly disabled as well.

See also section 2.7.2 “Dependency Chains“.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *speed* (unsigned 32bit), input - Sets the movement speed for closed-loop commands which is given in nanometers per second for linear positioners and in micro degrees per second for rotary positioners. The valid range is 0..100,000,000. A value of 0 (default) deactivates the speed control feature.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// move to zero position with 0,5 mm/s
result = SA_SetClosedLoopMoveSpeed_S(mcsHandle,0,500000);
result = SA_GotoPositionAbsolute_S(mcsHandle,0,0,0);
```

See also: `SA_GetClosedLoopMoveSpeed_S`, `SA_SetClosedLoopMoveAcceleration_S`, `SA_SetClosedLoopMaxFrequency_S`

SA_SetEndEffectorType_S

Channel type: End effector

Interface:

```
SA_STATUS SA_SetEndEffectorType_S(SA_INDEX systemIndex,
                                  SA_INDEX channelIndex,
                                  unsigned int type,
                                  signed int param1,
                                  signed int param2);
```

Description:

Each end effector channel must be configured with the type of end effector that is connected to it before it can be used. This function configures the type along with its parameters depending on the type. There are three types of end effectors:

- Gripper
- Force sensor
- Gripper with integrated force sensor

The parameters that the end effectors must be configured with are taken from the data sheets that come along with the end effectors.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *type* (unsigned 32bit), input - Specifies the type of the end effector. Possible values are `SA_GRIPPER_END_EFFECTOR_TYPE`, `SA_FORCE_SENSOR_END_EFFECTOR_TYPE` and `SA_FORCE_GRIPPER_END_EFFECTOR_TYPE`.
- *param1* (signed 32bit), input - The meaning and valid range of this parameter depends on the type given (see table below).
- *param2* (signed 32bit), input - The meaning and valid range of this parameter depends on the type given (see table below).

Type	param1		param2	
	Meaning	Valid Range	Meaning	Valid Range
Gripper (1)	maximum voltage [1/100 V]	100 – 22,500 (1 - 225V)	none	0
Force Sensor (2)	sensor gain [1/10 µN/V]	1 – 10,000 (0.1 – 1000.0 µN/V)	none	0
Force Gripper (3)	sensor gain [1/10 µN/V]	1 – 10,000 (0.1 – 1,000.0 µN/V)	maximum voltage [1/100 V]	100 – 22,500 (1 - 225V)

Example:

```
// configures channel three of the system to be a force gripper with a
// sensor gain of 42µN/V and a maximum gripper voltage of 150V.
result = SA_SetEndEffectorType_S(mcsHandle,3,
                                 SA_FORCE_GRIPPER_END_EFFECTOR_TYPE,420,15000);
```

See also: `SA_GetEndEffectorType_S`

SA_SetPosition_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetPosition_S(SA_INDEX systemIndex,
                           SA_INDEX channelIndex,
                           signed int position);
```

Description:

For positioners that have a sensor installed, this function may be used to define the current position resp. angle of the positioner to have a specific value. The measuring scale of the positioner is shifted accordingly.

If the positioner “knows” its physical position (via `SA_FindReferenceMark_S`) when calling this function, the resulting scale shift will be saved to non-volatile memory. On future power-ups it will recall its position automatically after calling `SA_FindReferenceMark_S`. See section 2.5.3 “Defining Positions” for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *position* (signed 32bit), input - Defines the value that the current position of the positioner should have. In case of a rotary positioner the range of this parameter is limited to 0 .. 359,999,999. Note that the revolution implicitly will always be set to 0.

Example:

```
// set the position of the first positioner connected to the system to 3.5mm
result = SA_SetPosition_S(mcsHandle,0,3500000);
```

See also: `SA_GetPhysicalPositionKnown_S`, `SA_FindReferenceMark_S`, `SA_SetScale_S`

SA_SetPositionLimit_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetPositionLimit_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                signed int minPosition,
                                signed int maxPosition);
```

Description:

For positioners with integrated sensors this function may be used to limit the travel range of a linear positioner by software. (For rotary positioners see `SA_SetAngleLimit_S`.) By default there is no limit set. If defined the positioner will not move beyond the limit. This affects open-loop as well as closed-loop movements.
Note that the limit may only be set if the physical position is known at the time of the call (see `SA_FindReferenceMark_S`, `SA_GetPhysicalPositionKnown_S`).

See section 2.5.4 “Software Range Limit“ for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *minPosition* (signed 32bit), input - Absolute minimum position given in nanometers.
- *maxPosition* (signed 32bit), input - Absolute maximum position given in nanometers.

Note: *maxPosition* must be greater than *minPosition*, otherwise the positioner will not move at all. If both parameters have the same value then the software range limit is disabled.

Example:

```
// limit the travel range of positioner 0 to +/- 3mm
result = SA_SetPositionLimit_S(mcsHandle,0,-3000000,3000000);
```

See also: `SA_GetPositionLimit_S`, `SA_SetAngleLimit_S`, `SA_FindReferenceMark_S`

SA_SetSafeDirection_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetSafeDirection_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                unsigned int direction);
```

Description:

Some sensor types are not equipped with a physical reference mark. For these positioners a mechanical end stop is used as a reference point when calling `SA_FindReferenceMark_S`. Which end stop is used is configured by the safe direction. 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 safe direction setting also affects the behavior of `SA_CalibrateSensor_S`. Positioners that are referenced via an end stop also move to the configured end stop as part of the calibration routine. Please be aware though that the calibration routine must not *start* near a mechanical end stop. Otherwise the calibration might fail and cause unexpected behavior in closed-loop mode.

The safe direction setting is a global parameter for a channel and affects `SA_CalibrateSensor_S` as well as `SA_FindReferenceMark_S`. Please note that latter function will ignore its direction parameter for positioners that are referenced via an end stop and will implicitly use the safe direction parameter instead.

This function has no effect on channels that have a sensor type configured that is referenced via a reference mark. See appendix 5.4 “Sensor Types“ in the appendix for a list of sensor types and their reference marks.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *direction* (unsigned 32bit), input - Sets the safe direction. Must be either `SA_FORWARD_DIRECTION` or `SA_BACKWARD_DIRECTION`.

Example:

```
SA_STATUS result = SA_SetSafeDirection_S(mcsHandle,1,SA_FORWARD_DIRECTION);
// The positioner will move in forward direction when referencing.
```

See also: `SA_GetSafeDirection_S`, `SA_CalibrateSensor_S`, `SA_FindReferenceMark_S`, `SA_SetSensorType_S`

SA_SetScale_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetScale_S(SA_INDEX systemIndex,
                        SA_INDEX channelIndex,
                        signed int scale,
                        unsigned int inverted);
```

Description:

This command may be used to the shift and invert the measuring scale of a positioner. Please see section 2.5.3 “Defining Positions“ for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *scale* (signed 32bit), input - Sets the desired scale shift relative to the physical scale of the positioner. The value is given in nano meters for linear positioners and in micro degrees for rotary positioners.
- *inverted* (unsigned 32bit) - Sets the scale inversion. Must be either `SA_FALSE` or `SA_TRUE`.

Caution: Please note that only the logical scale of the positioner will be inverted when the `inverted` value has changed. Parameters like the *SafeDirection* will not be altered. Thus the positioner will move in the opposite direction when e.g. calling SA_FindReferenceMark_S with the same parameters prior to the inversion change.

Example:

```
SA_STATUS result = SA_SetScale_S(mcsHandle,1,1000000,SA_FALSE);
// Sets the scale shift to +1mm relative to the physical scale.
```

See also: `SA_GetScale_S`, `SA_SetPosition_S`

SA_SetSensorEnabled_S

Interface:

```
SA_STATUS SA_SetSensorEnabled_S(SA_INDEX systemIndex,
                                unsigned int enabled);
```

Description:

This function may be used to activate or deactivate the sensors that are attached to the positioners of a system. The command is system global and affects all positioner channels of a system equally. It effectively turns the power supply of the sensors on or off. Please refer to section 2.5.2 “Sensor Modes” for more information on the sensor modes. End effector channels are not affected by this function.

If this command is issued, all positioner channels of the system are implicitly stopped.

This setting is stored to non-volatile memory immediately and need not be configured on every power-up.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *enabled* (unsigned 32bit), input - Sets the mode. Must be either `SA_SENSOR_DISABLED`, `SA_SENSOR_ENABLED` or `SA_SENSOR_POWERSAVE`.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// disable sensors
result = SA_SetSensorEnabled_S(mcsHandle,SA_SENSOR_DISABLED);
```

See also: `SA_GetSensorEnabled_S`

SA_SetSensorType_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetSensorType_S(SA_INDEX systemIndex,
                             SA_INDEX channelIndex,
                             unsigned int type);
```

Description:

When using positioners with integrated sensors, this function is used to tell a channel what type of positioner is connected. The type affects position calculation and functions that may be called for a channel (see for example `SA_GetPosition_S` and `SA_GetAngle_S`).

Please refer to appendix 5.4 “Sensor Types“ for a list of available sensor types.

Note that each channel stores this setting to non-volatile memory. Consequently, there is no need to call this function on every initialization. If the sensor type of a channel is changed, you must call `SA_CalibrateSensor_S` to ensure proper operation of the sensor.

If this command is issued, the positioner is implicitly stopped.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *type* (unsigned 32bit), input - Specifies the type of the sensor (see appendix 5.4 “Sensor Types“).

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// configure SR sensor for 2nd channel
result = SA_SetSensorType_S(mcsHandle,1,SA_SR_SENSOR_TYPE);
```

See also: `SA_GetSensorType_S`, `SA_GetPosition_S`, `SA_GetAngle_S`

SA_SetStepWhileScan_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetStepWhileScan_S(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                unsigned int step);
```

Description:

This function is of interest in conjunction with closed-loop commands (e.g. `SA_GotoPositionAbsolute_S`, see there) and sets a flag that affects the behavior of a positioner. If the positioner is instructed to hold the target position after reaching it, the scanning mode will primarily be used to hold the position. In this mode 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 function may be used to forbid the execution of steps even if this means that the position can not be held.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *step* (unsigned 32bit), input - Selects the mode. Must be either `SA_NO_STEP_WHILE_SCAN` or `SA_STEP_WHILE_SCAN`. The latter is the default.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
      // handle error...
}
// forbid to do correction steps while holding position
result = SA_SetStepWhileScan_S(mcsHandle,0,SA_NO_STEP_WHILE_SCAN);
```

See also: `SA_GotoPositionAbsolute_S`, `SA_GotoPositionRelative_S`, `SA_GotoAngleAbsolute_S`, `SA_GotoAngleRelative_S`

SA_SetZeroForce_S

Channel type: End effector

Interface:

```
SA_STATUS SA_SetZeroForce_S(SA_INDEX systemIndex,
                            SA_INDEX channelIndex);
```

Description:

End effectors that have a force sensor do not measure absolute force, but rather a change of force. For proper force measurement this function may be used to set the measured force to zero and should be called when the force sensor is mechanically unstressed.
Setting the zero force takes about one second. During this time the end effector will report a status of `SA_CALIBRATING_STATUS`. Note that in the asynchronous mode the channel will report completion of the command if configured so with the `SA_SetReportOnComplete_A` function (see there).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int status;
SA_SetZeroForce_S(0,1);
// wait until finished
do {
       SA_GetStatus_S(mcsHandle,1,&status);
} while (status != SA_STOPPED_STATUS);
// force sensor is set to zero
```

SA_StepMove_S

Channel type: Positioner

Interface:

```
SA_STATUS SA_StepMove_S(SA_INDEX systemIndex,
                        SA_INDEX channelIndex,
                        signed int steps,
                        unsigned int amplitude,
                        unsigned int frequency);
```

Description:

This is an open-loop command. It performs a burst of steps with the given parameters. Note that a single step is atomic. When interrupting a burst with `SA_Stop_S` the positioner will finish the current step before stopping. This implies that the piezo element of the positioner is always at its resting potential after a step command.

While executing the command the positioner will have a movement status of `SA_STEPPING_STATUS` (see `SA_GetStatus_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *steps* (signed 32bit), input - Number and direction of steps to perform. The valid range is -30,000..30,000. A value of 0 stops the positioner, but see `SA_Stop_S`. A value of 30,000 or -30,000 performs an unbounded move. This should be used with caution since the positioner will only stop on a `SA_Stop_S` command.
- *amplitude* (unsigned 32bit), input - Amplitude that the steps are performed with. Lower amplitude values result in a smaller step width. The parameter must be given as a 12bit value (range 0..4,095). 0 corresponds to 0V, 4,095 to 100V.
- *frequency* (unsigned 32bit), input - Frequency in Hz that the steps are performed with. The valid range is 1..18,500.

WARNING: It is strongly discouraged to use unbounded moves, especially at high frequencies! Positioners develop heat when they are driven and may take permanent damage if constantly driven over a long period of time (> 1 minute), especially in environments with weak thermal coupling. Be aware that if driven in the ultra sonic frequency range there is the risk of an unintended unbounded move to go unnoticed!

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       // handle error...
}
// perform 100 steps with full amplitude at 1kHz
result = SA_StepMove_S(mcsHandle,0,100,4095,1000);
```

SA_Stop_S

Channel type: Positioner, End effector

Interface:

```
SA_STATUS SA_Stop_S(SA_INDEX systemIndex,
                    SA_INDEX channelIndex);
```

Description:

Stops any ongoing movement of a positioner or end effector. Note that if a stepping movement is performed with a positioner the current step is completed before the positioner is stopped. This command also stops the hold position feature of closed-loop commands, such as `SA_GotoPositionAbsolute_S` or even `SA_FindReferenceMark_S`.

A positioner or end effector that is stopped will have a movement status of `SA_STOPPED_STATUS` (see `SA_GetStatus_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       // handle error...
}
// perform 1,000 steps with half amplitude at 1kHz
result = SA_StepMove_S(mcsHandle,0,1000,2048,1000);
// stop
result = SA_Stop_S(mcsHandle,0);
/*
Note: In this example the positioner will start executing 1,000 steps after the
SA_StepMove_S command. Since SA_Stop_S is called right away, the number of steps actually
executed (before the movement is aborted) is a timing issue and may depend on your PC
machine speed and/or the USB connection to the hardware.
*/
```

3.3 Functions for Asynchronous Communication

Most functions of the asynchronous communication mode have the same functionality as their counterparts of the synchronous mode. The main difference is the suffix of the function name. Please refer to the synchronous functions for details. However, there are a few functions that differ in their interface and also some additional functions which are described below.

All functions of the asynchronous communication mode are thread safe.

For functions that address a specific channel the channel type for which the function is callable is given.

SA_AppendTriggeredCommand_A

Channel type: Positioner

Interface:

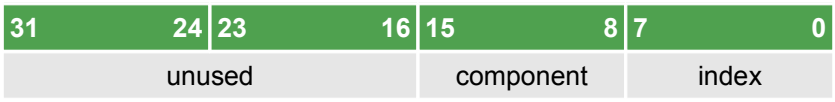
```
SA_STATUS SA_AppendTriggeredCommand_A(SA_INDEX systemIndex,
                                      SA_INDEX channelIndex,
                                      signed int triggerSource);
```

Description:

This function is used in combination with movement commands to fill the command queue with commands. Queued movement commands are not executed right away, but rather triggered by a configurable event source. Please refer to section 2.6.5 “Command Queues“ for more information.

After calling `SA_AppendTriggeredCommand_A`, the next movement command will be put into the command queue for later execution.

The *triggerSource* parameter must be given in form of a selector value which is a 32-bit code that refers to an event source and has the following structure:



It is recommended to use the `SA_ESV` helper function to encode the value. See also section 2.6 “Controller Event System“.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *triggerSource* (signed 32bit), input – Defines the event source that should be used to trigger the command. This parameter is coded in form of a selector value that refers to an event source.

Example:

```
// indicate to the channel that the next movement command should not be executed
// right away, but rather queued for later execution.
SA_STATUS result = SA_AppendTriggeredCommand_A(
       mcsHandle,                // system handle
       0,                        // channel index
       SA_ESV(SA_DIGITAL_IN,0)   // trigger source (selector value)
);
// now append the command to the queue
result = SA_GotoPositionAbsolute_A(mcsHandle,0,1000,0);
```

See also: `SA_ClearTriggeredCommandQueue_A`, `SA_SetReportOnTriggered_A`, `SA_TriggerCommand_A`

SA_CancelWaitForPacket_A

Interface:

```
SA_STATUS SA_CancelWaitForPacket_A(SA_INDEX systemIndex);
```

Description:

This function may be used in conjunction with `SA_LookAtNextPacket_A` or `SA_ReceiveNextPacket_A`. These functions block until a data packet is received from a system or a specified timeout occurs. Especially for an infinite timeout (`SA_TIMEOUT_INFINITE`) `SA_CancelWaitForPacket_A` may be called from a different thread to unblock the functions listed above and let them return an `SA_CANCELED_ERROR`.

Parameters:

- systemIndex (unsigned 32bit), input - Handle to an initialized system.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
//
SA_PACKET dataPacket;
SA_STATUS result;
result = SA_ReceiveNextPacket_A(mcsHandle,SA_TIMEOUT_INFINITE,&dataPacket);
switch (result) {
	case SA_OK:
		// handle received packet
		break;
	case SA_CANCELED_ERROR:
		// a different thread has called the SA_CancelWaitForPacket_A function to
		// manually unblock SA_ReceiveNextPacket_A above
		break;
	default:
		// handle other errors
		break;
}
```

See also: `SA_LookAtNextPacket_A`, `SA_ReceiveNextPacket_A`

SA_ClearTriggeredCommandQueue_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_ClearTriggeredCommandQueue_A(SA_INDEX systemIndex,
                                          SA_INDEX channelIndex);
```

Description:

This function is used in conjunction with command queuing (see section 2.6.5 “Command Queues“). If there are movement commands in the command queue then it is not possible to issue (non-queued) movement commands until the queue is empty again (all commands in the queue must have been triggered).

This function may be used to cancel all commands that are in the command queue. The commands will not be executed, but simply removed from the queue. After this the queue size will be zero.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.

Example:

```
// cancel all commands in the command queue
SA_STATUS result = SA_ClearTriggeredCommandQueue_A(mcsHandle,0);
```

See also: `SA_AppendTriggeredCommand_A`

SA_DiscardPacket_A

Interface:

```
SA_STATUS SA_DiscardPacket_A(SA_INDEX systemIndex);
```

Description:

This function can be used to discard a received data packet. If a data packet was received earlier, but not consumed (removed from the receive buffer) it may be dropped by this function. If the receive buffer is empty, this function has no effect. See section 2.3.3 “Retrieving Answers” for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.

Example:

```
// request position of positioner
result = SA_GetPosition_A(mcsHandle,0);
// poll until answer is there
SA_PACKET packet;
packet.packetType = SA_NO_PACKET_TYPE;
while (packet.packetType != SA_POSITION_PACKET_TYPE) {
       result = SA_LookAtNextPacket_A(mcsHandle,1000,&packet);
       if (packet.packetType != SA_POSITION_PACKET_TYPE)
              SA_DiscardPacket_A(mcsHandle);
}
```

See also: `SA_ReceiveNextPacket_A`, `SA_LookAtNextPacket_A`

SA_FlushOutput_A

Interface:

```
SA_STATUS SA_FlushOutput_A(SA_INDEX systemIndex);
```

Description:

When buffered output is used (see `SA_SetBufferedOutput_A`) this function may be used to initiate the transmission of the commands that are stored in the output buffer. After this the buffer of the given system is cleared.

Please note that when using buffered output this function should be called regularly. If too many commands are accumulated in the buffer an `SA_OUTPUT_BUFFER_OVERFLOW_ERROR` will be returned by the function that caused the buffer to overflow.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.

Example: see example of `SA_SetBufferedOutput_A`

See also: `SA_SetBufferedOutput_A`

SA_GetAngle_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetAngle_A(SA_INDEX systemIndex,
                        SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the current positioner angle. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

This command 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error. Additionally, the command is only executable if the addressed channel is configured to be of type rotary (see `SA_SetSensorType_S`). A linear channel will return an error (use `SA_GetPosition_A` instead).

The addressed channel will reply with a packet of type `SA_ANGLE_PACKET_TYPE`. The *data1* field will hold the positioner angle and the *data2* field will hold the positioner revolution.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current angle
result = SA_GetAngle_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_ANGLE_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// packet.data1 holds current angle
		// packet.data2 holds current revolution
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_GetAngle_S`, `SA_GetPosition_S`

SA_GetAngleLimit_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetAngleLimit_A(SA_INDEX systemIndex,
                             SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the travel range limit that is currently configured for a rotary positioner. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_ANGLE_LIMIT_PACKET_TYPE`. The *data1* field will hold the minimum angle, the *data2* field will hold the minimum revolution, the *data4* field will hold the maximum angle and the *data3* field will hold the maximum revolution.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
// request range limit
result = SA_GetAngleLimit_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_ANGLE_LIMIT_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// minimum angle is in packet.data1
		// minimum revolution is in packet.data2
		// maximum angle is in packet.data4
		// maximum revolution is in packet.data3
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_GetAngleLimit_S`

SA_GetBufferedOutput_A

Interface:

```
SA_STATUS SA_GetBufferedOutput_A(SA_INDEX systemIndex,
                                 unsigned int *mode);
```

Description:

Inverse function to `SA_SetBufferedOutput_A` (see there). Returns the current output buffer mode.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *mode* (unsigned 32bit), output – Holds the current buffer mode. Will be either `SA_UNBUFFERED_OUTPUT` or `SA_BUFFERED_OUTPUT`.

Example:

```
// check current buffer mode
unsigned int mode;
result = SA_GetBufferedOutput_A(mcsHandle,&mode);
if (result == SA_OK) {
       // mode holds the current buffer mode
}
```

See also: `SA_SetBufferedOutput_A`

SA_GetCaptureBuffer_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetCaptureBuffer_A(SA_INDEX systemIndex,
                                SA_INDEX channelIndex,
                                unsigned int bufferIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the contents of a capture buffer. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_CAPTURE_BUFFER_PACKET_TYPE`. The *data1* field will always contain the buffer index that the packet represents (same as given in the *bufferIndex* parameter when requesting the packet). The values of the other fields hold the capture buffer contents (please see `SA_GetCaptureBuffer_S`).

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *bufferIndex* (unsigned 32bit): input – Selects the buffer.

Example:

```
// request capture buffer 0
SA_STATUS result = SA_GetCaptureBuffer_A(mcsHandle,0,0);
// receive answer
SA_PACKET packet;
result = SA_ReceiveNextPacket_A(mcsHandle,0,1000,&packet);
```

See also: `SA_GetCaptureBuffer_S`

SA_GetChannelProperty_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetChannelProperty_A(SA_INDEX systemIndex,
                                  SA_INDEX channelIndex,
                                  unsigned int key);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the value of a property. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The property which is to be read is selected via the key parameter. This 32-bit parameter codes a combination of values and has the following structure:

31 24	23 16	15 8	7 0
component	sub component	property	

The `SA_EPK` helper function may be used to encode the key. See the header file for a list of component selectors and properties. The sub component selector is usually an index, but there can also be special sub component selectors. Note that not all properties are valid for all components. Please refer to section 2.4 “Channel Properties“ for more information.

The addressed channel will reply with a packet of type `SA_CHANNEL_PROPERTY_PACKET_TYPE`. The *data1* field will contain the property key that the packet represents (same as given in the *key* parameter when requesting the packet). The *data2* field will contain the value of the property.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.
- *key* (unsigned 32bit), input – Specifies the property key from which the value should be read.

Example:

```
// request command queue capacity
SA_STATUS result = SA_GetChannelProperty_A(mcsHandle,
                                           0,
                                           SA_EPK(SA_COMMAND_QUEUE,0,SA_CAPACITY));
// receive answer
SA_PACKET packet;
result = SA_ReceiveNextPacket_A(mcsHandle,0,1000,&packet);
if (packet.packetType == SA_CHANNEL_PROPERTY_PACKET_TYPE) {
      // queue capacity is in packet.data2
}
```

See also: `SA_SetChannelProperty_S`

SA_GetClosedLoopMoveAcceleration_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetClosedLoopMoveAcceleration_A(SA_INDEX systemIndex,
                                             SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the movement acceleration that is currently configured for the channel (see `SA_SetClosedLoopMoveAcceleration_S`). The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_MOVE_ACCELERATION_PACKET_TYPE`. The *data1* field will hold the movement acceleration. The unit is µm/s² for linear positioners and in m°/s² for rotary positioners.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input – Selects the channel of the selected system. The index is zero based.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
// request currently configured acceleration
SA_STATUS result = SA_GetClosedLoopMoveAcceleration_A(mcsHandle,0);
// receive answer
SA_PACKET packet;
result = SA_ReceiveNextPacket_A(mcsHandle,0,1000,&packet);
if (packet.packetType == SA_MOVE_ACCELERATION_PACKET_TYPE) {
      // acceleration is in packet.data1
}
```

See also: `SA_SetClosedLoopMoveAcceleration_S`

SA_GetClosedLoopMoveSpeed_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetClosedLoopMoveSpeed_A(SA_INDEX systemIndex,
                                      SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the movement speed that is currently configured for the channel (see `SA_SetClosedLoopMoveSpeed_S`). The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_MOVE_SPEED_PACKET_TYPE`. The *data1* field will hold the movement speed in nm/s for linear positioner and in µ°/s for rotary positioners.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Note: This Command is not available on all controllers. Please contact SmarAct for more information.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current movement speed
result = SA_GetClosedLoopMoveSpeed_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_MOVE_SPEED_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// movement speed is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_SetClosedLoopMoveSpeed_S`

SA_GetEndEffectorType_A

Channel type: End effector

Interface:

```
SA_STATUS SA_GetEndEffectorType_A(SA_INDEX systemIndex,
                                  SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the end effector type that is currently configured for the channel (see `SA_SetEndEffectorType_S`). The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_END_EFFECTOR_TYPE_PACKET_TYPE`. The *data1* field will hold the end effector type. The fields *data2* and *data3* will hold the type parameters. See `SA_SetEndEffectorType_S` for more information on these parameters.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current end effector type
result = SA_GetEndEffectorType_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_END_EFFECTOR_TYPE_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// end effector type is in packet.data1
		// param1 is in packet.data2
		// param2 is in packet.data3
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_SetEndEffectorType_S`

SA_GetForce_A

Channel type: End effector

Interface:

```
SA_STATUS SA_GetForce_A(SA_INDEX systemIndex,
                        SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the force that is currently measured by the force sensor. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_FORCE_PACKET_TYPE`. The *data2* field will hold the force given in 1/10 µN.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current force
result = SA_GetForce_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_FORCE_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// force is in packet.data2
	} else {
		// handle packet otherwise
	}
}
```

SA_GetGripperOpening_A

Channel type: End effector

Interface:

```
SA_STATUS SA_GetGripperOpening_A(SA_INDEX systemIndex,
                                 SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the voltage that is currently applied to the gripper. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_GRIPPER_OPENING_PACKET_TYPE`. The *data1* field will hold the voltage given in 1/100 Volts.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current gripper position
result = SA_GetGripperOpening_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_GRIPPER_OPENING_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// voltage is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_GotoGripperOpeningAbsolute_S`, `SA_GotoGripperOpeningRelative_S`

SA_GetPhysicalPositionKnown_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetPhysicalPositionKnown_A(SA_INDEX systemIndex,
                                        SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return whether the positioner “knows” its physical position or not. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_PHYSICAL_POSITION_KNOWN_PACKET_TYPE`. The *data1* field will hold the known-status.

See also 2.5.3 “Defining Positions“.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
// request known-status
result = SA_GetPhysicalPositionKnown_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_PHYSICAL_POSITION_KNOWN_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// known-status is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_GetPhysicalPositionKnown_S`

SA_GetPosition_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetPosition_A(SA_INDEX systemIndex,
                           SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the current position of the positioner. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

This command 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 `SA_SetSensorEnabled_S`). If this is not the case the channel will return an error. Additionally, the command is only executable if the addressed channel is configured to be of type linear (see `SA_SetSensorType_S`). A rotary channel will return an error (use `SA_GetAngle_A` instead).

The addressed channel will reply with a packet of type `SA_POSITION_PACKET_TYPE`. The *data2* field will hold the current position.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
       // handle error...
}
// request current position
result = SA_GetPosition_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
       // handle error
} else {
       if ((packet.packetType == SA_POSITION_PACKET_TYPE) &&
              (packet.channelIndex == 0)) {
              // current position is in packet.data2
       } else {
              // handle packet otherwise
       }
}
```

See also: `SA_GetPosition_S`, `SA_GetAngle_S`

SA_GetPositionLimit_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetPositionLimit_A(SA_INDEX systemIndex,
                                SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the travel range limit that is currently configured for a linear positioner. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_POSITION_LIMIT_PACKET_TYPE`. The *data2* field will hold the minimum position and the *data3* field will hold the maximum position.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
// request range limit
result = SA_GetPositionLimit_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_POSITION_LIMIT_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// minimum position is in packet.data2
		// maximum position is in packet.data3
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_GetPositionLimit_S`

SA_GetSafeDirection_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetStatus_A(SA_INDEX systemIndex,
                         SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the currently configured safe direction. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_SAFE_DIRECTION_PACKET_TYPE`. The *data1* field will hold the current safe direction.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current status
result = SA_GetSafeDirection_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_SAFE_DIRECTION_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// safe direction is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

SA_GetScale_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetScale_A(SA_INDEX systemIndex,
                        SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the scale shift and inversion that is currently configured for a linear positioner. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_SCALE_PACKET_TYPE`. The *data2* field will hold the scale shift.

Please see section 2.5.3 “Defining Positions“ for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Caution: Please note that only the logical scale of the positioner will be inverted when the `inverted` value has changed. Parameters like the *SafeDirection* will not be altered. Thus the positioner will move in the opposite direction when e.g. calling SA_FindReferenceMark_S with the same parameters prior to the inversion change.

Example:

```
// request scale shift
result = SA_GetScale_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
       // handle error
} else {
       if ((packet.packetType == SA_SCALE_PACKET_TYPE) &&
              (packet.channelIndex == 0)) {
              // scale shift is in packet.data2
              // inversion is in packet.data1
       } else {
              // handle packet otherwise
       }
}
```

See also: `SA_GetScale_S, SA_SetSafeDirection_S`

SA_GetSensorEnabled_A

Interface:

```
SA_STATUS SA_GetSensorEnabled_A(SA_INDEX systemIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a system to return the current sensor mode (see `SA_SetSensorEnabled_S`). The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed system will reply with a packet of type `SA_SENSOR_ENABLED_PACKET_TYPE`. The *data1* field will hold the sensor mode.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current sensor mode
result = SA_GetSensorEnabled_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mscHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if (packet.packetType == SA_SENSOR_ENABLED_PACKET_TYPE) {
		// sensor mode is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_SetSensorEnabled_S`

SA_GetSensorType_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetSensorType_A(SA_INDEX systemIndex,
                             SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the sensor type that is currently configured for the channel (see `SA_SetSensorType_S`). The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_SENSOR_TYPE_PACKET_TYPE`. The *data1* field will hold the sensor type.

Please refer to `SA_SetSensorType_S` for more information on the sensor types.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current sensor type
result = SA_GetSensorType_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_SENSOR_TYPE_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// sensor type is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

See also: `SA_SetSensorType_S`

SA_GetStatus_A

Channel type: Positioner, End effector

Interface:

```
SA_STATUS SA_GetStatus_A(SA_INDEX systemIndex,
                         SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the current positioner movement status. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_STATUS_PACKET_TYPE`. The *data1* field will hold the current movement status of the positioner.

Please refer to appendix 5.3 “Channel Status Codes“ for a list of status codes.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
	// handle error...
}
// request current status
result = SA_GetStatus_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
	// handle error
} else {
	if ((packet.packetType == SA_STATUS_PACKET_TYPE) &&
		(packet.channelIndex == 0)) {
		// positioner movement status code is in packet.data1
	} else {
		// handle packet otherwise
	}
}
```

SA_GetVoltageLevel_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_GetVoltageLevel_A(SA_INDEX systemIndex,
                               SA_INDEX channelIndex);
```

Description:

In contrast to the synchronous version of this function, this function sends a query to a channel to return the voltage level that is currently applied to the piezo element. The actual answer must be retrieved via other functions, e.g. `SA_ReceiveNextPacket_A`.

The addressed channel will reply with a packet of type `SA_VOLTAGE_LEVEL_PACKET_TYPE`. The *data1* field will hold the voltage level.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
       // handle error...
}
// request current voltage level
result = SA_GetVoltageLevel_A(mcsHandle,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(mcsHandle,1000,&packet);
if (result != SA_OK) {
       // handle error
} else {
       if ((packet.packetType == SA_VOLTAGE_LEVEL_PACKET_TYPE) &&
              (packet.channelIndex == 0)) {
              // voltage level is in packet.data1
       } else {
              // handle packet otherwise
       }
}
```

See also: `SA_ScanMoveAbsolute_S`, `SA_ScanMoveRelative_S`

SA_LookAtNextPacket_A

Interface:

```
SA_STATUS SA_LookAtNextPacket_A(SA_INDEX systemIndex,
                                unsigned int timeout,
                                SA_PACKET *packet);
```

Description:

This function may be used to receive a data packet from the MCS without actually consuming it. Depending on the timeout parameter the function will block or not (see below). If no packet was received then a packet of type `SA_NO_PACKET_TYPE` is returned. If a packet is received then it is returned, but not removed from the receive buffer. The packet may be "looked at" as often as desired. It is consumed by either calling `SA_ReceiveNextPacket_A` or `SA_DiscardPacket_A` (see there). See section 2.3.3 "Retrieving Answers" for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *timeout* (unsigned 32bit), input - Specifies how long to wait for incoming data. If no data is received after timeout milli seconds the function will return with a packet type of `SA_NO_PACKET_TYPE`. A value of `SA_TIMEOUT_INFINITE` will only let the function return when a packet is received. In this case the function may be unblocked manually by `SA_CancelWaitForPacket_A`.
- *packet* (`SA_PACKET`), output - If the call was successful this value holds the received data packet. Depending on the packet type the various fields of the packet are valid or not. See the appendix for a reference.

Example:

```
// request position of positioner
SA_GetPosition_A(mcsHandle,0);
// wait until answer is there
SA_PACKET packet;
SA_LookAtNextPacket_A(mcsHandle,1000,&packet);
SA_DiscardPacket_A();
if ((packet.packetType == SA_POSITION_PACKET_TYPE) && (packet.channelIndex == 0)) {
      // position is in packet.data2
}
```

See also: `SA_ReceiveNextPacket_A`, `SA_DiscardPacket_A`, `SA_CancelWaitForPacket_A`

SA_ReceiveNextPacket_A

Interface:

```
SA_STATUS SA_ReceiveNextPacket_A(SA_INDEX systemIndex,
                                 unsigned int timeout,
                                 SA_PACKET *packet);
```

Description:

This function may be used to receive a data packet from the MCS. Depending on the timeout parameter the function will block or not (see below). If a packet is received it is returned and consumed by the call. If no packet was received a packet of type `SA_NO_PACKET_TYPE` is returned. See section 2.3.3 “Retrieving Answers” for more information.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *timeout* (unsigned 32bit), input - Specifies how long to wait for incoming data. If no data is received after timeout milli seconds the function will return with a packet type of `SA_NO_PACKET_TYPE`. A value of `SA_TIMEOUT_INFINITE` will only let the function return when a packet is received. In this case the function may be unblocked manually by `SA_CancelWaitForPacket_A`.
- *packet* (`SA_PACKET`), output - If the call was successful this value holds the received data packet. Depending on the packet type the various fields of the packet are valid or not. See the appendix for a reference.

Example:

```
unsigned int mcsHandle;
const char loc[] = “usb:id:3118167233”;
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, “async”);
if (result != SA_OK) {
       // handle error...
}
// request position of positioner
result = SA_GetPosition_A(msHandle,0);
// receive answer, but don't wait longer than five seconds
SA_PACKET packet;
result = SA_ReceiveNextPacket_A(mcsHandle,5000,&packet);
```

See also: `SA_LookAtNextPacket_A`, `SA_DiscardPacket_A`, `SA_CancelWaitForPacket_A`

SA_SetBufferedOutput_A

Interface:

```
SA_STATUS SA_SetBufferedOutput_A(SA_INDEX systemIndex,
                                 unsigned int mode);
```

Description:

This function may be used to optimize the communication with the hardware in asynchronous communication mode. It selects between one of two modes that affect the way commands are sent.

- Unbuffered – This is the default mode. In this mode commands are sent to the hardware as soon as you call its function. When calling multiple functions in fast succession there may be a delay between them due to the way the underlying USB communication works.
- Buffered – In this mode commands are not sent to the hardware immediately. Instead, the data is held back and stored in an internal buffer. You may accumulate several commands and then call `SA_FlushOutput_A` to initiate the transmission of the stored commands.

Note that each system has a separate output buffer.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *mode* (unsigned 32bit), input - Selects the mode. Must be either `SA_UNBUFFERED_OUTPUT` or `SA_BUFFERED_OUTPUT`.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
       // handle error...
}
// set buffered output mode
SA_SetBufferedOutput_A(mcsHandle,SA_BUFFERED_OUTPUT);
// send movement commands to channels
SA_GotoPositionRelative_A(mcsHandle,0,1000000,0);    // channel 0: move 1mm forward
SA_GotoPositionRelative_A(mcsHandle,1,-1000000,0);   // channel 1: move 1mm backward
// up until here no movement is executed.
// now flush the buffer
SA_FlushOutput_A(mcsHandle);
// now both movement commands are executed.
```

See also: `SA_GetBufferedOutput_A`, `SA_FlushOutput_A`

SA_SetReportOnComplete_A

Channel type: Positioner, End effector

Interface:

```
SA_STATUS SA_SetReportOnComplete_A(SA_INDEX systemIndex,
                                   SA_INDEX channelIndex,
                                   unsigned int report);
```

Description:

This function tells a channel whether or not to report the completion of the last movement command. If set to true, the channel will send a packet of type `SA_COMPLETED_PACKET_TYPE` when it has completed the movement. See section ?? “Report on Complete” for more information. The default behavior is no reporting.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *report* (unsigned 32bit), input - Must be `SA_DISABLED` or `SA_ENABLED`.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
      // handle error...
}
result = SA_SetReportOnComplete_A(mcsHandle,0,SA_ENABLED);
result = SA_GotoPositionAbsolute_A(mcsHandle,0,1000,0);
// positioner is moving. wait for completion
SA_PACKET packet;
result = SA_ReceiveNextPacket_A(mcsHandle,                //this call blocks
                                SA_TIMEOUT_INFINITE,      // until a packet
                                &packet);                 // is received
if (result != SA_OK) {
      // handle error
}
if ((packet.packetType == SA_COMPLETED_PACKET_TYPE) && (packet.channelIndex == 0)) {
      // movement command has completed
}
```

SA_SetReportOnTriggered_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_SetReportOnTriggered_A(SA_INDEX systemIndex,
                                    SA_INDEX channelIndex,
                                    unsigned int report);
```

Description:

This function tells a channel whether or not to report when a movement command from the command queue has been triggered (see section 2.6.5 “Command Queues“ for more information). If set to true, the channel will send a packet of type `SA_TRIGGERED_PACKET_TYPE` when the next command in the queue has been triggered. The default behavior is no reporting.

Parameters:

- *systemIndex* (unsigned 32bit), input - Handle to an initialized system.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *report* (unsigned 32bit), input - Must be `SA_DISABLED` or `SA_ENABLED`.

Example:

```
unsigned int mcsHandle;
const char loc[] = “usb:id:3118167233”;
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, “async”);
if (result != SA_OK) {
       // handle error...
}
result = SA_SetReportOnTriggered_A(mscHandle,0,SA_ENABLED);
```

See also: `SA_AppendTriggeredCommand_A`, `SA_TriggerCommand_A`

SA_TriggerCommand_A

Channel type: Positioner

Interface:

```
SA_STATUS SA_TriggerCommand_A(SA_INDEX systemIndex,
                              unsigned int triggerIndex);
```

Description:

This command triggers commands that were loaded into the command queue with `SA_AppendTriggeredCommand_A` (see chapter 2.6.5 “Command Queues“) while specifying the software trigger as event source. Note that this command is global to a system and is sent to all channels of a system simultaneously. However, the command will only be triggered if the index that was given while specifying the event source is the same as the *triggerIndex* passed to `SA_TriggerCommand_A`. This mechanism enables you to e.g. preload all channels of a system with movement commands, but group different sets of channels that should start their movement at different times.

Parameters:

- *systemIndex* (unsigned 32bit), input – Handle to an initialized system.
- triggerIndex (unsigned 32bit), input – Index of the trigger command. Only commands that were loaded with this index are actually triggered. The valid range is 0 .. 255.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "async");
if (result != SA_OK) {
      // handle error...
}
result = SA_AppendTriggeredCommand_A(mcsHandle,0,SA_ESV(SA_SOFTWARE_TRIGGER,12));
// next movement command will be queued and triggered by software, index 12
result = SA_GotoPositionAbsolute_A(mcsHandle,0,10000,0);
// command is queued. positioner is not moving yet
result = SA_TriggerCommand_A(mcsHandle,12);   // index must be same as above
// command is now in execution
```

See also: `SA_AppendTriggeredCommand_A`

3.4 Miscellaneous Functions

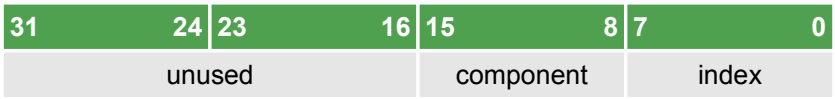
SA_DSV

Interface:

```
void SA_DSV(signed int value,
            unsigned int *selector,
            unsigned int *subSelector);
```

Description:

This function (Decode Selector Value) is used in conjunction with `SA_GetChannelProperty_S`. Some component property values represent a selector value. Selector values are references to other components and have the following structure:



When reading selector values `SA_GetChannelProperty_S` returns a 32-bit value which may be decoded and split up into its fields with this function.

See also chapter 2.4 “Channel Properties“.

Parameters:

- *value* (signed 32bit), input – The value returned by `SA_GetChannelProperty_S` to be decoded.
- *selector* (unsigned 32bit), output – The decoded selector value referring to a component.
- *subSelector* (unsigned 32bit), output – The decoded sub selector value referring to a sub component.

Example:

```
// read trigger source of counter 0
signed int value;
SA_STATUS result = SA_GetChannelProperty_S(
      mcsHandle,                                  // system handle
      0,                                          // channel index
      SA_EPK(SA_COUNTER,0,SA_TRIGGER_SOURCE),     // property key
      &value                                      // property value
);
unsigned int selector, subSelector;
SA_DSV(value,&selector,&subSelector);
// selector refers to the component, e.g. SA_DIGITAL_IN
// subSelector refers to the sub component, e.g. 0 (index)
```

See also: `SA_ESV`, `SA_GetChannelProperty_S`

SA_EPK

Interface:

```
unsigned int SA_EPK(unsigned int selector,
                    unsigned int subSelector,
                    unsigned int property);
```

Description:

This function (Encode Property Key) is used in conjunction with `SA_GetChannelProperty_S` and `SA_SetChannelProperty_S`. The property which is to be read or written is selected via the key parameter. This 32-bit parameter codes a combination of values and has the following structure:

31 24	23 16	15 8	7 0
component	sub component	property	

The `SA_EPK` helper function may be used to encode the key. See the header file for a list of component selectors and properties. The sub component selector is usually an index, but there can also be special sub component selectors. Note that not all properties are valid for all components. Please refer to section 2.4 “Channel Properties“ for more information.

Parameters:

- *selector* (unsigned 32bit), input – The component that should be referenced to.
- *subSelector* (unsigned 32bit), input – The sub component that should be referenced to.
- property (unsigned 32bit), input – The property of the component.

Returns the encoded property key.

Example:

```
// set trigger source of counter 0 to digital in 0
SA_STATUS result = SA_SetChannelProperty_S(
      mcsHandle,                                  // system handle
      0,                                          // channel index
      SA_EPK(SA_COUNTER,0,SA_TRIGGER_SOURCE),     // property key
      SA_ESV(SA_DIGITAL_IN,0)                     // property value (selector)
);
```

See also: `SA_GetChannelProperty_S`, `SA_SetChannelProperty_S`

SA_ESV

Interface:

```
signed int SA_ESV(unsigned int selector,
                  unsigned int subSelector);
```

Description:

This function (Encode Selector Value) is used in conjunction with `SA_SetChannelProperty_S`. Some component property values represent a selector value. Selector values are references to other components and have the following structure:

31 24	23 16	15 8	7 0
unused		component	index

When setting selector values `SA_SetChannelProperty_S` expects a value that codes a reference to another component which may be encoded with this function.

See also chapter 2.4 “Channel Properties“.

Parameters:

- *selector* (unsigned 32bit), input – The component that should be referenced to.
- *subSelector* (unsigned 32bit), input – The sub component that should be referenced to.

Returns the encoded value that may be passed directly to `SA_SetChannelProperty_S`.

Example:

```
// set trigger source of counter 0 to digital in 0
SA_STATUS result = SA_SetChannelProperty_S(
       mcsHandle,                                  // system handle
       0,                                          // channel index
       SA_EPK(SA_COUNTER,0,SA_TRIGGER_SOURCE),     // property key
       SA_ESV(SA_DIGITAL_IN,0)                     // selector value
);
```

See also: `SA_DSV`, `SA_SetChannelProperty_S`

SA_GetStatusInfo

Interface:

```
SA_STATUS SA_GetStatusInfo(SA_STATUS status,
                           const char **info);
```

Description:

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

Please refer to appendix 5.1 “Function Status Codes“ for a list of function status codes.

Parameters:

- *status* (unsigned 32bit), input – A status code that was returned by a library function call.
- *info* (pointer to char), output – Pointer to a null terminated string that describes the error code.

Example:

```
unsigned int mcsHandle;
const char loc[] = "usb:id:3118167233";
SA_STATUS result = SA_OpenSystem(&mcsHandle, loc, "sync");
if (result != SA_OK) {
       const char *errorText
       SA_GetStatusInfo(result,&errorText);
       // errorText points to a descriptive text
}
```

4 Quick Reference

This section gives an overview of the available functions along with a brief description. They are grouped by their general purpose and each group is sorted alphabetically.

4.1 Initialization Functions

4.2 Configuration Functions

4.3 Movement Control Functions

4.4 Channel Feedback Functions

4.5 Answer Retrieval Functions

4.6 Miscellaneous Functions

5 Appendix

5.1 Function Status Codes

All library functions return a status code that indicates success or failure of execution. This chapter lists all possible status codes and gives a short description.

Code	Symbol Description
0	SA_OK The function call was successful.
1	SA_INITIALIZATION_ERROR An error occurred while initializing the library. All systems should be disconnected and reset before the next attempt is made.
2	SA_NOT_INITIALIZED_ERROR A function call has been made for an uninitialized system. Call SA_OpenSystem before communicating with the hardware.
3	SA_NO_SYSTEMS_FOUND_ERROR May occur at initialization if no Modular Control Systems have been detected on the PC system. Check the connection of the USB cable and make sure the drivers are installed properly. Note: After power-up / USB connection it may take several seconds for the system to be detectable.
4	SA_TOO_MANY_SYSTEMS_ERROR The number of allowed systems connected to the PC is limited to 32. If you have more connected, disconnect some.
5	SA_INVALID_SYSTEM_INDEX_ERROR An invalid system index has been passed to a function. The system index parameter of various functions is zero based. If N is the number of acquired systems, then the valid range for the system index is 0..(N-1).
6	SA_INVALID_CHANNEL_INDEX_ERROR An invalid channel index has been passed to a function. The channel index parameter of various functions is zero based. If N is the number of channels available on a system, then the valid range for the channel index is 0..(N-1).
7	SA_TRANSMIT_ERROR An error occurred while sending data to the hardware. The system should be reset.
8	SA_WRITE_ERROR An error occurred while sending data to the hardware. The system should be reset.
9	SA_INVALID_PARAMETER_ERROR An invalid parameter has been passed to a function. Check the function documentation for valid ranges.
10	SA_READ_ERROR An error occurred while receiving data from the hardware. The system should be reset.
12	SA_INTERNAL_ERROR An internal communication error occurred. The system should be reset.
13	SA_WRONG_MODE_ERROR The called function does not match the communication mode that was selected at initialization (see 2.2.2 – “Communication Modes“). In synchronous communication mode only functions of sections I and IIa may be called. In asynchronous communication mode only functions of sections I and IIb may be called.
14	SA_PROTOCOL_ERROR An internal protocol error occurred. The system should be reset.

15 SA_TIMEOUT_ERROR
The hardware did not respond. Make sure that all cables are connected properly and reset the system.

17 SA_ID_LIST_TOO_SMALL_ERROR
When calling SA_GetAvailableSystems you must pass a pointer to an array that is large enough to hold the system IDs of all connected systems. If the number of detected systems is larger than the array, this error will be generated.

18 SA_SYSTEM_ALREADY_ADDED_ERROR
In order to acquire specific systems you must call SA_AddSystemToInitSystemsList before calling SA_InitSystems. A system ID may only be added once to the list of systems to be acquired. Multiple calls with the same ID lead to this error.

19 SA_WRONG_CHANNEL_TYPE_ERROR
Most functions of section II are only callable for certain channel types. For example, calling SA_StepMove_S for a channel that is an end effector channel will lead to this error. The detailed function description notes the types of channels that the function may be called for.

20 SA_CANCELED_ERROR
The functions SA_ReceiveNextPacket_A and SA_LookAtNextPacket_A return this code if they were blocking while waiting for an incoming packet and then were manually unblocked by calling SA_CancelWaitForPacket_A.

21 SA_INVALID_SYSTEM_LOCATOR_ERROR
Returned by SA_OpenSystem if the locator string does not comply with the supported locator formats.

22 SA_INPUT_BUFFER_OVERFLOW_ERROR
This error occurs when the input buffer for storing packets that have been received from a system is full. To avoid this error remove packets from the input buffer frequently with SA_ReceiveNextPacket_A.

23 SA_QUERYBUFFER_SIZE_ERROR
Returned by functions that write data in a binary or char buffer (e.g. SA_FindSystems) if the user-supplied buffer is too small to hold the returned data.

24 SA_DRIVER_ERROR
Returned by functions, if a driver, that is required to communicate to a controller over a certain hardware interface, is not available.

129 SA_NO_SENSOR_PRESENT_ERROR
This error occurs if a function was called that requires sensor feedback, but the addressed positioner has none attached.

130 SA_AMPLITUDE_TOO_LOW_ERROR
The amplitude parameter that was given is too low.

131 SA_AMPLITUDE_TOO_HIGH_ERROR
The amplitude parameter that was given is too high.

132 SA_FREQUENCY_TOO_LOW_ERROR
The frequency parameter that was given is too low.

133 SA_FREQUENCY_TOO_HIGH_ERROR
The frequency parameter that was given is too high.

135 SA_SCAN_TARGET_TOO_HIGH_ERROR
The target position for a scanning movement that was given is too high.

136 SA_SCAN_SPEED_TOO_LOW_ERROR
The scan speed parameter that was given for a scan movement command is too low.

137 SA_SCAN_SPEED_TOO_HIGH_ERROR

The scan speed parameter that was given for a scan movement command is too high.

140 SA_SENSOR_DISABLED_ERROR
This error occurs if an addressed positioner has a sensor attached, but it is disabled. See SA_SetSensorEnabled_S.

141 SA_COMMAND_OVERRIDEN_ERROR
This error is only generated in the asynchronous communication mode. When the software commands a movement which is then interrupted by the Hand Control Module, an error of this type is generated.

142 SA_END_STOP_REACHED_ERROR
This error is generated by a positioner channel in asynchronous mode if the target position of a closed-loop command could not be reached because a mechanical end stop was detected. After this error the positioner will have the SA_STOPPED_STATUS status code.

143 SA_WRONG_SENSOR_TYPE_ERROR
This error occurs if a closed-loop command does not match the sensor type that is currently configured for the addressed channel. For example, calling SA_GetPosition_S while the targeted channel is configured as rotary will lead to this error.

144 SA_COULD_NOT_FIND_REF_ERROR
This error is generated in asynchronous mode if the search for a reference mark was aborted. See section ?? “Reference Marks” for more information.

145 SA_WRONG_END_EFFECTOR_TYPE_ERROR
This error occurs if a command does not match the end effector type that is currently configured for the addressed channel. For example, calling SA_GetForce_S while the targeted channel is configured for a gripper will lead to this error.

146 SA_MOVEMENT_LOCKED_ERROR
Generated either if a movement command was aborted due to an emergency stop or a movement command was issued while the channel is in the locked state. See 2.4.1 “Emergency Stop“.

147 SA_RANGE_LIMIT_REACHED_ERROR
If a range limit is defined by SA_SetPositionLimit_A or SA_SetAngleLimit_A and the positioner is about to move beyond this limit, then the positioner will stop and report this error. After this error the positioner will have the SA_STOPPED_STATUS status code.

148 SA_PHYSICAL_POSITION_UNKNOWN_ERROR
A range limit is only allowed to be defined if the positioner “knows” its physical position. If this is not the case, the functions SA_SetPositionLimit_X and SA_SetAngleLimit_X will return this error code.

149 SA_OUTPUT_BUFFER_OVERFLOW_ERROR
When using buffered output in asynchronous communication mode, this error is returned if too many commands were accumulated in the output buffer. Call SA_FlushOutput_A to prevent this. See SA_SetBufferedOutput_A for more information.

150 SA_COMMAND_NOT_PROCESSABLE_ERROR
This error is generated if a command is sent to a channel when it is in a state where the command cannot be processed. For example, to change the sensor type of a channel the addressed channel must be completely stopped. In this case send a stop command before changing the type.

151 SA_WAITING_FOR_TRIGGER_ERROR
If there is at least one command queued in the command queue then you may only append more commands (if the queue is not full), but you may not issue movement commands for immediate execution. Doing so will generate this error. See section 2.6.5 "Command Queues".

152 SA_COMMAND_NOT_TRIGGERABLE_ERROR
After calling SA_AppendTriggeredCommand_A you are required to issue a movement command that is to be triggered by the given event source. Commands that cannot be triggered will generate this error.

153 SA_COMMAND_QUEUE_FULL_ERROR

This error is generated if you attempt to append more commands to the command queue, but the queue cannot hold anymore commands. The queue capacity may be read out with `SA_GetChannelProperty_S` (see there).

154 `SA_INVALID_COMPONENT_ERROR`
Indicates that a component was selected that does not exist.

155 `SA_INVALID_SUB_COMPONENT_ERROR`
Indicates that a sub component was selected that does not exist.

156 `SA_INVALID_PROPERTY_ERROR`
Indicates that the selected component does not have the selected property.

157 `SA_PERMISSION_DENIED_ERROR`
This error is generated when you call a functionality which is not unlocked for the system (e.g. Low Vibration Mode).

161 `SA_INCOMPLETE_PACKET_ERROR`
This error is generated by the hardware if a packet was not received completely and thus a timeout occurs.

164 `SA_RX_BUFFER_OVERFLOW_ERROR`
The system could not process all incoming command packets and an internal buffer overflow occures. This indicates to a potential misuse of the programming API. Too many commands were send without reading the returned answer packets. The system went to an error state where further communication is blocked until the system is restarted.

240 `SA_UNKNOWN_COMMAND_ERROR`
This error occurs if the library sends a command that is not supported by the system it is sent to. This may be the case when using a newer library with an older firmware version. Please be sure only to use the library that is shipped with your system to avoid compatibility problems.

255 `SA_OTHER_ERROR`
An error that can't be otherwise categorized.

5.2 Packet Types

Note: The packet type determines which fields of the packet hold valid values. It is therefore advised to first check the type of the packet before making any further checks.

Code	Symbol Description	Valid fields
0	SA_NO_PACKET_TYPE A packet of this type does not represent an actual data packet. It simply indicates that no packet was received. None of the other fields are valid.	None
1	SA_ERROR_PACKET_TYPE If a command could not be executed or some other error occurred, an error is generated. The channelIndex field holds the source channel and the data1 field the error code (see listing above).	channelIndex, data1
2	SA_POSITION_PACKET_TYPE This packet type results from a SA_GetPosition_A function call. The channelIndex holds the source channel and the data2 field holds the current position in nano meters.	channelIndex, data2
3	SA_COMPLETED_PACKET_TYPE If a channel has been configured to report the completion of a movement command, a packet of this type is generated on this event (see SA_SetReportOnComplete_A). The channelIndex field holds the source channel.	channelIndex
4	SA_STATUS_PACKET_TYPE This packet type results from a SA_GetStatus_S function call. The channelIndex holds the source channel and the data1 field holds the current movement status code (see listing below).	channelIndex, data1
5	SA_ANGLE_PACKET_TYPE This packet type results from a SA_GetAngle_A function call. The channelIndex holds the source channel, the data1 field holds the angle in micro degrees and the data2 field holds the revolution.	channelIndex, data1, data2
6	SA_VOLTAGE_LEVEL_PACKET_TYPE This packet type results from a SA_GetVoltageLevel_A function call. The channelIndex holds the source channel and the data1 field holds the current voltage level that is applied to the piezo element of the positioner. The returned value ranges from 0..4,095. A 0 corresponds to 0V, a 4,095 to 100V.	channelIndex, data1
7	SA_SENSOR_TYPE_PACKET_TYPE This packet type results from a SA_GetSensorType_A function call. The channelIndex holds the source channel and the data1 field holds the sensor type, which will be SA_S_SENSOR_TYPE, SA_SR_SENSOR_TYPE, SA_ML_SENSOR_TYPE, SA_MR_SENSOR_TYPE or SA_SP_SENSOR_TYPE. If the connected positioner is not equipped with a sensor, SA_NO_SENSOR_TYPE will be returned.	channelIndex, data1
8	SA_SENSOR_ENABLED_PACKET_TYPE This packet type results from a SA_GetSensorEnabled_A function call. Since the answer is system global, the channelIndex is not defined in packets of this type. The data1 field holds the currently configured sensor mode and will be one of SA_SENSOR_DISABLED, SA_SENSOR_ENABLED or SA_SENSOR_POWERSAVE.	channelIndex, data1
9	SA_END_EFFECTOR_TYPE_PACKET_TYPE This packet type results from a SA_GetEndEffectorType_A function call. The channelIndex holds the source channel and the data1 field holds the currently configured end effector type, which will be SA_ANALOG_SENSOR_END_EFFECTOR_TYPE, SA_GRIPPER_END_EFFECTOR_TYPE, SA_FORCE_SENSOR_END_EFFECTOR_TYPE or SA_FORCE_GRIPPER_END_EFFECTOR_TYPE. The fields data2 and data3 hold the type parameters, which depend on the end effector type.	channelIndex, data1, data2, data3
10	SA_GRIPPER_OPENING_PACKET_TYPE This packet type results from a SA_GetGripperOpening_A function call. The channelIndex holds the source channel and the data1 field holds the voltage given in 1/100 Volts.	channelIndex, data1
11	SA_FORCE_PACKET_TYPE This packet type results from a SA_GetForce_A function call. The channelIndex holds the source channel and the data2 field holds the force given in 1/10 µN.	channelIndex, data2

12 SA_MOVE_SPEED_PACKET_TYPE channelIndex, data1
This packet type results from a SA_GetClosedLoopMoveSpeed_A function call. The channelIndex holds the source channel and the data1 field holds the movement speed. The unit is nm/s for linear positioners and µ°/s for rotary positioners.

13 SA_PHYSICAL_POSITION_KNOWN_PACKET_TYPE channelIndex, data1
This packet type results from a SA_GetPhysicalPositionKnown_A function call. The channelIndex holds the source channel. The data1 field will be either SA_PHYSICAL_POSITION_UNKNOWN or SA_PHYSICAL_POSITION_KNOWN.

14 SA_POSITION_LIMIT_PACKET_TYPE channelIndex, data2, data3
This packet type results from a SA_GetPositionLimit_A function call. The channelIndex holds the source channel, the data2 field holds the minimum position and the data3 field holds the maximum position.

15 SA_ANGLE_LIMIT_PACKET_TYPE channelIndex, data1, data2, data3, data4
This packet type results from a SA_GetAngleLimit_A function call. The channelIndex holds the source channel, the data1 field holds the minimum angle, the data2 field holds the minimum revolution, the data4 field holds the maximum angle and the data3 field holds the maximum revolution.

16 SA_SAFE_DIRECTION_PACKET_TYPE channelIndex, data1
This packet type results from a SA_GetSafeDirection_A function call. The channelIndex holds the source channel and the data1 field holds the currently configured safe direction of the channel.

17 SA_SCALE_PACKET_TYPE channelIndex, data2
This packet type results from a SA_GetScale_A function call. The channelIndex holds the source channel and the data2 field holds the currently configured scale shift of the channel.

18 SA_MOVE_ACCELERATION_PACKET_TYPE channelIndex, data1
This packet type results from a SA_GetClosedLoopMoveAcceleration_A function call. The channelIndex holds the source channel and the data1 field holds the currently configured movement acceleration. The unit is µm/s² for linear positioners and m°/s² for rotary positioners.

19 SA_CHANNEL_PROPERTY_PACKET_TYPE channelIndex, data1, data2
This packet type results from a SA_GetChannelProperty_A function call. The channelIndex holds the source channel, the data1 field holds the property key and the data2 field holds the property value.

20 SA_CAPTURE_BUFFER_PACKET_TYPE channelIndex, data1 (data2, data3, data4)
This packet type results from a SA_GetCaptureBuffer_A function call. The channelIndex holds the source channel and the data1 field holds the capture buffer index. The value of the other data fields depend on which capture buffer was retrieved. Please refer to the table at SA_GetCaptureBuffer_S.

21 SA_TRIGGERED_PACKET_TYPE channelIndex
If a channel has been configured to report whether a movement command has been triggered, a packet of this type is generated on this event (see SA_SetReportOnTriggered_A). The channelIndex field holds the source channel.

255 SA_INVALID_PACKET_TYPE
This dummy type indicates that the received packet is not a valid packet.

5.3 Channel Status Codes

The table below lists the status codes of positioners or end effectors that are returned by `SA_GetStatus_S`.

Code	Symbol Description
0	`SA_STOPPED_STATUS` The positioner or end effector is currently not performing active movement.
1	`SA_STEPPING_STATUS` The positioner is currently performing an open-loop movement (see `SA_StepMove_S`).
2	`SA_SCANNING_STATUS` The positioner is currently performing a scanning movement (e.g. `SA_ScanMoveAbsolute_S`).
3	`SA_HOLDING_STATUS` The positioner or end effector is holding its current target (see closed-loop commands, e.g. `SA_GotoPositionAbsolute_S`, `SA_GotoAngleAbsolute_S`, `SA_GotoGripperForceAbsolute_S`) or is holding the reference mark (see `SA_FindReferenceMark_S`).
4	`SA_TARGET_STATUS` The positioner or end effector is currently performing a closed-loop movement.
5	`SA_MOVE_DELAY_STATUS` The positioner is currently waiting for the sensors to power-up before executing the movement command. This status may be returned if the the sensors are operated in power save mode.
6	`SA_CALIBRATING_STATUS` The positioner or end effector is busy calibrating its sensor.
7	`SA_FINDING_REF_STATUS` The positioner is moving to find the reference mark.
8	`SA_OPENING_STATUS` The end effector (gripper) is closing or opening its jaws.

5.4 Sensor Types

The following table lists the currently available sensor types that may be configured with `SA_SetSensorType_S` (pass the type code when calling the function, see also the header file of the library for definitions).

The reference type indicates the way the positioner is referenced when calling `SA_FindReferenceMark_S`. Positioners with 'mark' are referenced via a physical reference mark that is typically located near the middle of the complete travel range. Positioners with 'end stop' are referenced via a mechanical end stop (see `SA_SetSafeDirection_S` for more information). Positioners with 'none' cannot be referenced.

Symbol	*Type Code*	*Positioner Series*	*Comment*	*Reference Type*
S	1	SLCxxxxs	linear positioner with nano sensor	mark
SR	2	SR36xxs, SR3511s, SR5714s, SR7021s, SR2812s	rotary positioner with nano sensor	mark
SP	5	SLCxxxxrs	linear positioner with nano sensor, large actuator	mark
SC	6	SLCxxxxsc	linear positioner with nano sensor, distance coded reference marks	mark*
SR20	8	SR2013s, SR1612s	rotary positioner with nano sensor	mark
M	9	SLCxxxxm	linear positioner with micro sensor	end stop
GD	11	SGO60.5m	goniometer with micro sensor (60.5mm radius)	end stop
GE	12	SGO77.5m	goniometer with micro sensor (77.5mm radius)	end stop
GF	14	SR1209m	rotary positioner with micro sensor	end stop
G605S	16	SGO60.5s	goniometer with nano sensor (60.5mm radius)	mark
G775S	17	SGO77.5s, SGO70.5s	goniometer with nano sensor (77.5mm or 70.5mm radius)	mark
SC500	18	SLLxxsc	linear positioner with nano sensor, distance coded reference marks	mark*
G955S	19	SGO95.5s	goniometer with nano sensor (95.5mm radius)	mark
SR77	20	SR77xxs	rotary positioner with nano sensor	mark
SD	21	SLCxxxxds, SLLxxs	like S, but with extended scanning Range	mark
R20ME	22	SR2013sx, SR1410sx	rotary positioner with micro sensor	mark
SR2	23	SR36xxs, SR3511s, SR5714s, SR7021s, SR2812s	like SR, for high applied masses	mark
SCD	24	SLCxxxxdsc	like SP, but with distance coded reference marks	mark*
SRC	25	SR7021sc	like SR, but with distance coded reference marks	mark*
SR36M	26	SR3610m	rotary positioner, no end stops	none
SR36ME	27	SR3610m	rotary positioner with end stops	end stop
SR50M	28	SR5018m	rotary positioner, no end stops	none
SR50ME	29	SR5018m	rotary positioner with end stops	end stop
G1045S	30	SGO104.5s	goniometer with nano sensor (104.5mm radius)	mark
G1395S	31	SGO139.5s	goniometer with nano sensor (139.5mm radius)	mark
MD	32	SLCxxxxdme	like M, but with large actuator	end stop
G935M	33	SGO93.5me	Goniometer with micro sensor (93.5mm radius)	end stop
SHL20	34	SHL-20	High load vertical positioner	mark
SCT	35	SLCxxxxscu	like SCD, but with even larger actuator	mark*

* These positioners are equipped with multiple reference marks. The positioner will only have to move a few millimeters to know its physical position.

5.5 Channel Properties

The table below lists the valid component, sub component and property combinations. If a component has an index as sub component then the table entry lists the currently available index values. Some component properties are read-only which is indicated in the “Access” column. These keys may only be passed to `SA_GetChannelProperty_S`, but not to `SA_SetChannelProperty_S`.

Component	*Sub Component*	*Property*	*Access*	*Valid Value Range*	*Page*
SA_GENERAL	SA_EMERGENCY_STOP	SA_OPERATION_MODE	R / W	SA_ESM_NORMAL*, SA_ESM_RESTRICTED, SA_ESM_DISABLED, SA_ESM_AUTO_RELEASE	12
		SA_DEFAULT_OPERATION_MODE	R / W	SA_ESM_NORMAL*, SA_ESM_RESTRICTED, SA_ESM_DISABLED, SA_ESM_AUTO_RELEASE	12
	SA_LOW_VIBRATION ([1])	SA_OPERATION_MODE	R / W	SA_DISABLED*, SA_ENABLED	13
	SA_BROADCAST_STOP	SA_OPERATION_MODE	R / W	SA_DISABLED*, SA_ENABLED	14
	SA_POSITION_CONTROL	SA_FORCED_SLIP	R / W	SA_DISABLED, SA_ENABLED*	14
SA_SENSOR	SA_POWER_SUPPLY	SA_OPERATION_MODE	R / W	SA_DISABLED, SA_ENABLED, SA_POWERSAVE*	14
	SA_SCALE	SA_OFFSET	R / W	±2,000,000,000	14
		SA_INVERTED	R / W	SA_FALSE*, SA_TRUE	14

Controller Event System related Channel Properties

Component	*Sub Component*	*Property*	*Access*	*Valid Value Range*	*Page*
SA_DIGITAL_IN	0	SA_OPERATION_MODE	R / W	SA_DISABLED*, SA_ENABLED	23
		SA_ACTIVE_EDGE	R / W	SA_FALLING_EDGE*, SA_RISING_EDGE	
SA_COUNTER	0	SA_TRIGGER_SOURCE	R / W	SA_DISABLED*, <Selector Value>	23
		SA_VALUE	R / W	0* .. 2,147,486,647	
SA_CAPTURE_BUFFER	0	SA_TRIGGER_SOURCE	R / W	SA_DISABLED*, <Selector Value>	24
SA_COMMAND_QUEUE	0	SA_SIZE	R	0* .. n	24
		SA_CAPACITY	R	n	
SA_SOFTWARE_TRIGGER ([2])	0 .. 255	none	N/A	N/A	23

* indicates default values of the properties

([1]) This feature is not available on all controllers. Please contact SmarAct for more information.

([2]) This component has no properties and may only be used as a trigger source.