



## ***MCS - Modular Control System Programmer's Guide***

# Table of Contents

|  |    |
|--|----|
| 1 Introduction.....                              | 4  |
| 2 Overview.....                                  | 4  |
| 2.1 Initialization.....                          | 5  |
| 2.1.1 Acquiring Specific Systems.....            | 5  |
| 2.1.2 Communication Modes.....                   | 5  |
| 2.2 Overwriting Movement Commands.....           | 5  |
| 3 Detailed Function Description.....             | 6  |
| 3.1 Initialization Functions.....                | 6  |
| SA_GetDLLVersion.....                            | 6  |
| SA_GetAvailableSystems.....                      | 7  |
| SA_AddSystemToInitSystemsList.....               | 8  |
| SA_ClearInitSystemsList.....                     | 9  |
| SA_InitSystems.....                              | 10 |
| SA_ReleaseSystems.....                           | 11 |
| SA_GetNumberOfSystems.....                       | 12 |
| SA_GetSystemID.....                              | 13 |
| SA_GetNumberOfChannels.....                      | 14 |
| SA_GetChannelType.....                           | 15 |
| SA_SetHCMEEnabled.....                           | 16 |
| 3.2 Functions for Synchronous Communication..... | 17 |
| SA_SetClosedLoopMaxFrequency_S.....              | 17 |
| SA_SetClosedLoopMoveSpeed_S.....                 | 18 |
| SA_GetClosedLoopMoveSpeed_S.....                 | 19 |
| SA_SetPosition_S.....                            | 20 |
| SA_SetZeroPosition_S.....                        | 21 |
| SA_GetPhysicalPositionKnown_S.....               | 22 |
| SA_SetPositionLimit_S.....                       | 23 |
| SA_GetPositionLimit_S.....                       | 24 |
| SA_SetAngleLimit_S.....                          | 25 |
| SA_GetAngleLimit_S.....                          | 26 |
| SA_SetStepWhileScan_S.....                       | 27 |
| SA_SetSensorEnabled_S.....                       | 28 |
| SA_GetSensorEnabled_S.....                       | 29 |
| SA_SetSensorType_S.....                          | 30 |
| SA_GetSensorType_S.....                          | 31 |
| SA_SetAccumulateRelativePositions_S.....         | 32 |
| SA_SetEndEffectorType_S.....                     | 33 |
| SA_GetEndEffectorType_S.....                     | 34 |
| SA_SetZeroForce_S.....                           | 35 |
| SA_StepMove_S.....                               | 36 |
| SA_ScanMoveAbsolute_S.....                       | 37 |
| SA_ScanMoveRelative_S.....                       | 38 |
| SA_GotoPositionAbsolute_S.....                   | 39 |
| SA_GotoPositionRelative_S.....                   | 40 |
| SA_GotoAngleAbsolute_S.....                      | 41 |
| SA_GotoAngleRelative_S.....                      | 42 |
| SA_Stop_S.....                                   | 43 |
| SA_CalibrateSensor_S.....                        | 44 |
| SA_FindReferenceMark_S.....                      | 45 |
| SA_GotoGripperOpeningAbsolute_S.....             | 46 |
| SA_GotoGripperOpeningRelative_S.....             | 47 |
| SA_GotoGripperForceAbsolute_S.....               | 48 |
| SA_GetVoltageLevel_S.....                        | 49 |
| SA_GetPosition_S.....                            | 50 |
| SA_GetAngle_S.....                               | 51 |
| SA_GetStatus_S.....                              | 52 |
| SA_GetGripperOpening_S.....                      | 53 |

|   |    |
|---|----|
| SA_GetForce_S.....                                | 54 |
| 3.3 Functions for Asynchronous Communication..... | 55 |
| SA_GetClosedLoopMoveSpeed_A.....                  | 55 |
| SA_GetPhysicalPositionKnown_A.....                | 56 |
| SA_GetPositionLimit_A.....                        | 57 |
| SA_GetAngleLimit_A.....                           | 58 |
| SA_SetReportOnComplete_A.....                     | 59 |
| SA_GetSensorEnabled_A.....                        | 60 |
| SA_GetSensorType_A.....                           | 61 |
| SA_GetEndEffectorType_A.....                      | 62 |
| SA_GetVoltageLevel_A.....                         | 63 |
| SA_GetPosition_A.....                             | 64 |
| SA_GetAngle_A.....                                | 65 |
| SA_GetStatus_A.....                               | 66 |
| SA_GetGripperOpening_A.....                       | 67 |
| SA_GetForce_A.....                                | 68 |
| SA_SetReceiveNotification_A.....                  | 69 |
| SA_ReceiveNextPacket_A.....                       | 70 |
| SA_ReceiveNextPacketIfChannel_A.....              | 71 |
| SA_LookAtNextPacket_A.....                        | 72 |
| SA_DiscardPacket_A.....                           | 73 |
| 4 Using the Asynchronous Mode.....                | 74 |
| 4.1 Overview.....                                 | 74 |
| 4.2 Sending Commands.....                         | 74 |
| 4.3 Retrieving Answers.....                       | 75 |
| 4.3.1 Data Packet Format.....                     | 75 |
| 4.3.2 Event Driven Answer Retrieval.....          | 76 |
| 4.4 Multi Threaded Applications.....              | 76 |
| 4.4.1 Thread Termination.....                     | 77 |
| 4.5 Other Issues.....                             | 77 |
| 4.5.1 Report on Complete.....                     | 77 |
| 4.5.2 Multiple Command Sources.....               | 78 |
| 5 Working With Sensor Feedback.....               | 79 |
| 5.1 Rotary Sensors.....                           | 79 |
| 5.2 Sensor Modes.....                             | 79 |
| 5.3 Defining Positions.....                       | 80 |
| 5.3.1 Reference Marks.....                        | 80 |
| 5.3.2 Recalling Positions.....                    | 82 |
| 5.4 Software Range Limit.....                     | 83 |
| 6 Appendix.....                                   | 85 |
| 6.1 Status / Error Codes.....                     | 85 |
| 6.2 Packet Types.....                             | 88 |
| 6.3 Channel Status Codes.....                     | 90 |

# 1 Introduction

This document describes the usage of the MCSControl.dll which is used to control one or more MCS systems 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 DLL summarizes the functions of the DLL and lists definitions like error and status codes.

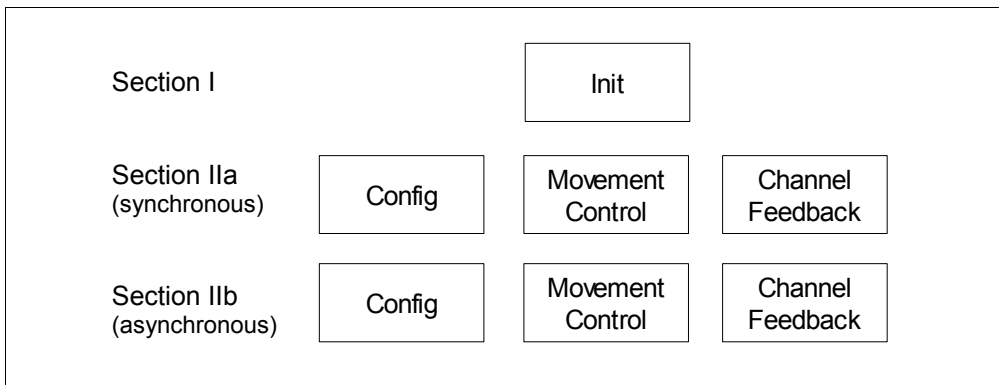
You may connect several MCS systems to your PC. Each of these is referred to as a “system” in the following. Most function calls require a system index as parameter to address a specific system. The system indexes are zero based. When using only one system the system index must always be zero.

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.

# 2 Overview

The functions of the DLL 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.



The figure above depicts the structure of the function groups. Section II (a or b) is again divided into several sub sections. The configuration functions may be used to configure the channels of a system, the movement control functions are used to tell a positioner or end effector connected to a channel what to do and the channel feedback functions mainly retrieve information from the channels.

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 DLL 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 the appendix for a complete list of status 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 was not `SA_OK`. This may result in unwanted behavior if these values are further processed.

For simplicity, all function parameters are 32 bits wide, either signed or unsigned. Note though that most parameters have a limited valid range. See the 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.

## **2.1 Initialization**

Before being able to use the hardware you must establish a connection to the system(s). This is done by the global initialization function `SA_InitSystems`. After this the functions of section II are “unlocked” and may be used to communicate with the hardware. Furthermore, the functions `SA_GetNumberOfSystems`, `SA_GetSystemID`, `SA_GetNumberOfChannels` and `SA_GetChannelType` may be used to retrieve information about the systems that were acquired.

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_ReleaseSystems` before it is free to be used by other applications.

### **2.1.1 Acquiring Specific Systems**

Normally, when calling `SA_InitSystems`, all systems that are connected to the PC at the time of the call are acquired. However, it is also possible to acquire specific systems. This may be useful for example if you have several systems connected to one PC, but want to control them separately with independent applications.

Each system has a unique system ID that is labeled on the housing of the device and can also be read out by software (see below). To acquire one or more specific systems you must supply their system IDs to the DLL before calling `SA_InitSystems`. The DLL manages a list of system IDs that are to be acquired. Initially, this list is empty. In this special case all available systems are acquired which is the default behavior. You may use the functions `SA_AddSystemToInitSystemsList` and `SA_ClearInitSystemsList` to manipulate the system ID list. After having filled the list, the `SA_InitSystems` function will try to acquire the given systems. It is advised though to check whether the desired systems have actually been acquired by using the functions `SA_GetNumberOfSystems` and `SA_GetSystemID`.

The `SA_GetAvailableSystems` function may be used to retrieve a list of detected systems that are ready for acquisition. However, you are not required to use this function. If for example an application is to be bound to a specific device of which the ID is already known, you may add its system ID directly to the initialization list.

### **2.1.2 Communication Modes**

When calling `SA_InitSystems` you must chose between two communication modes. The different modes affect internal communication but also the way you must use the DLL. 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 DLL handles the answer retrieval.

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. See section 4 “Using the Asynchronous Mode” for more information.

## **2.2 Overwriting Movement Commands**

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

However, a special situation occurs when overwriting movement commands while using the report-on-complete-feature in the asynchronous mode. See section 4.5.1 “Report on Complete” for more information.

## 3 Detailed Function Description

### 3.1 Initialization Functions

#### SA\_GetDLLVersion

**Interface:**

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

**Description:**

This function may be called to retrieve the version code of the DLL. 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 DLL behaves as the application expects it to do.

The returned 32bit code is divided into three fields:

|      |        |               |
|------|--------|---------------|
| Bits | 31..24 | Version High  |
| Bits | 16..23 | Version Low   |
| Bits | 0..15  | Version Build |

This function does not require the DLL to be initialized (see `SA_InitSystems`) 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\_GetAvailableSystems

## 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.

See section 2.1.1 “Acquiring Specific Systems” for more information.

## 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\_AddSystemToInitSystemsList

### Interface:

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

### Description:

This function may be used to acquire specific systems at initialization. The DLL 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.

See section 2.1.1 “Acquiring Specific Systems” for more information.

### 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\_ClearInitSystemsList

### 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.

See section 2.1.1 “Acquiring Specific Systems” for more information.

### 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\_InitSystems

## 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 DLL 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_GetAvailableSystems`, `SA_AddSystemToInitSystemsList` and `SA_ClearInitSystemsList`.

If the DLL is already initialized an implicit `SA_ReleaseSystems` call is made before (re-)initializing the DLL. 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. See section 2.1.1 "Acquiring Specific Systems" for more information.

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\_ReleaseSystems**

### **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\_GetNumberOfSystems

### 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

### 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\_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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
unsigned int number;  
result = SA_GetNumberOfChannels(0, &number);  
if (result == SA_OK) {  
    // number holds the number of channels of the first system  
}
```

# 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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
unsigned int type;  
result = SA_GetChannelType(0,0,&type);  
if (result == SA_OK) {  
    // type holds the channel type of the first channel of the first system  
}
```

# 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 4.5.2 "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 - Selects the system. The index is zero based.
- **enabled** (unsigned 32bit), input - Selects the mode. Must be one of SA\_HCM\_DISABLED, SA\_HCM\_ENABLED or SA\_HCM\_CONTROLS\_DISABLED.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// disable the Hand Control Module  
result = SA_SetHCMEnabled(0, 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\_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 - Selects the system. The index is zero based.
- *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:**

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// set maximum closed-loop frequency of first positioner connected to the first  
// system to 3kHz.  
result = SA_SetClosedLoopMaxFrequency_S(0,0,3000);
```

# 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 sensor of a positioner should be calibrated for proper operation of the speed control (see `SA_CalibrateSensor_S`).

Please also note that the speed control feature is currently only implemented for linear positioners.

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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. The valid range is 0..100,000,000. A value of 0 (default) deactivates the speed control feature.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// move to zero position with 0,5 mm/s  
result = SA_SetClosedLoopMoveSpeed_S(0,0,500000);  
result = SA_GotoPositionAbsolute_S(0,0,0,0);
```

**See also:** `SA_GetClosedLoopMoveSpeed_S`, `SA_SetClosedLoopMaxFrequency_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 - Selects the system. The index is zero based.
- *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 and has a valid range of 0..100,000,000.

### Example:

```
// get current movement speed for channel 2 of system 1  
unsigned int speed;  
result = SA_GetClosedLoopMoveSpeed_S(1,2,&speed);  
if (result == SA_OK) {  
    // speed holds the current movement speed  
}
```

**See also:** `SA_SetClosedLoopMoveSpeed_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. This function replaces `SA_SetZeroPosition_S`.

If the positioner “knows” its physical position (via `SA_FindReferenceMark_S`) when calling this function, the position will be saved to non-volatile memory. On future power-ups it will recall its physical position automatically after calling `SA_FindReferenceMark_S`.

See section 5.3 “Defining Positions” for more information.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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 first system to 3.5mm  
result = SA_SetPosition_S(0,0,3500000);
```

**See also:** `SA_GetPhysicalPositionKnown_S`, `SA_FindReferenceMark_S`

# SA\_SetZeroPosition\_S

**Channel type:** Positioner

## Interface:

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

## Description:

For positioners that have a sensor installed, this function may be used to define the current position resp. angle of the positioner as the zero position resp. angle.

Please note that this function is deprecated. Call `SA_SetPosition_S` instead.

## Parameters:

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

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// reset position of the first positioner connected to the first system  
result = SA_SetZeroPosition_S(0,0);
```

**See also:** `SA_SetPosition_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 5.3 “Defining Positions”.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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 of system 1 is known  
unsigned int known;  
result = SA_GetPhysicalPositionKnown_S(1,2,&known);  
if (result == SA_OK) {  
    // known holds the result  
}
```

**See also:** `SA_FindReferenceMark_S`, `SA_SetPosition_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`).

See section 5.4 “Software Range Limit“ for more information.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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(0,0,-3000000,3000000);
```

**See also:** `SA_GetPositionLimit_S`, `SA_SetAngleLimit_S`, `SA_FindReferenceMark_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 - Selects the system. The index is zero based.
- *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(0,0,&min,&max);  
if (result == SA_OK) {  
    // min and max hold the range limit  
}
```

**See also:** `SA_SetPositionLimit_S`, `SA_GetAngleLimit_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`).

See section 5.1 “Rotary Sensors“ and 5.4 “Software Range Limit“ for more information.

**Parameters:**

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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(0,0,350000000,-1,1000000,0);
```

**See also:** `SA_GetAngleLimit_S`, `SA_SetPositionLimit_S`, `SA_FindReferenceMark_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 - Selects the system. The index is zero based.
- *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(0,0,&min,&minR,&max,&maxR);
```

**See also:** `SA_SetAngleLimit_S`, `SA_GetPositionLimit_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 - Selects the system. The index is zero based.
- `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.

### Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// forbid to do correction steps while holding position  
result = SA_SetStepWhileScan_S(0,0,SA_NO_STEP_WHILE_SCAN);
```

**See also:** `SA_GotoPositionAbsolute_S`, `SA_GotoPositionRelative_S`,  
`SA_GotoAngleAbsolute_S`, `SA_GotoAngleRelative_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 5.2 “Sensor Modes” for more information on the sensor modes.

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

Note that if the system is reset, then the sensor mode after the reset depends on the default setting of the system. If a Hand Control Module is attached, the default setting may be changed there. If you utilize the sensor mode feature, it is therefore recommended to call this function after initialization or check the mode with `SA_GetSensorEnabled_S`.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *enabled* (unsigned 32bit), input - Sets the mode. Must be either `SA_SENSOR_DISABLED`, `SA_SENSOR_ENABLED` or `SA_SENSOR_POWERSAVE`.

### Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// disable sensors  
result = SA_SetSensorEnabled_S(0, SA_SENSOR_DISABLED);
```

**See also:** `SA_GetSensorEnabled_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 set 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 5.2 “Sensor Modes” for more information on the sensor modes.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// read sensor mode  
unsigned int enabled;  
result = SA_GetSensorEnabled_S(0, &enabled);
```

**See also:** SA\_SetSensorEnabled\_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 may be 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`).

Note that each channel stores this setting to non-volatile memory. Consequently, there is no need to call this function on every initialization.

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

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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. Must be one of `SA_S_SENSOR_TYPE`, `SA_SR_SENSOR_TYPE`, `SA_ML_SENSOR_TYPE`, `SA_MR_SENSOR_TYPE`, `SA_SP_SENSOR_TYPE` or `SA_SC_SENSOR_TYPE`.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// configure rotary sensor for 2nd channel of first system  
result = SA_SetSensorType_S(0,1,SA_ROTARY_SENSOR_TYPE);
```

**See also:** `SA_GetSensorType_S`, `SA_GetPosition_S`, `SA_GetAngle_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`). Usually, the returned type will be either `SA_S_SENSOR_TYPE`, `SA_SR_SENSOR_TYPE`, `SA_ML_SENSOR_TYPE`, `SA_MR_SENSOR_TYPE` or `SA_SP_SENSOR_TYPE`. A special case occurs if the positioner of the addressed channel has no sensor attached to it. In this case a sensor type of `SA_NO_SENSOR_TYPE` will be returned. Therefore, this function may be used to detect if a sensor is present or not.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get sensor type for 2nd channel of first system  
unsigned int sensorType;  
result = SA_GetSensorType_S(0,1,&sensorType);
```

**See also:** `SA_SetSensorType_S`, `SA_GetPosition_S`, `SA_GetAngle_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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_GetVoltageLevel_A(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// disable accumulation of relative movement commands in closed-loop mode  
result = SA_SetAccumulateRelativePositions_S(0,0,SA_NO_ACCUMULATE_RELATIVE_POSITIONS);
```

**See also:** `SA_GotoPositionRelative_S`, `SA_GotoAngleRelative_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 four types of end effectors:

- Analog sensor
- Gripper
- Force sensor
- Gripper with integrated force sensor

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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_ANALOG_SENSOR_END_EFFECTOR_TYPE`, `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.
- *param2* (signed 32bit), input - The meaning and valid range of this parameter depends on the type given.

## Example:

**See also:** `SA_GetEndEffectorType_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 - Selects the system. The index is zero based.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *type* (unsigned 32bit), input - If the call was successful this parameter holds the type of end effector that is currently configured for the channel. Possible values are SA\_ANALOG\_SENSOR\_END\_EFFECTOR\_TYPE, 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.
- *param2* (signed 32bit), input - The meaning and valid range of this parameter depends on the type.

### Example:

**See also:** SA\_GetEndEffectorType\_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 - Selects the system. The index is zero based.
- *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(0,1,&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:

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 - Selects the system. The index is zero based.
- *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.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// perform 100 steps with full amplitude at 1kHz  
result = SA_StepMove_S(0,0,100,4095,1000);
```

# 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 - Selects the system. The index is zero based.
- *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.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// scan to 50V  
result = SA_ScanMoveAbsolute_S(0,0,2048,10000);  
// scan to 0V within two seconds  
result = SA_ScanMoveAbsolute_S(0,0,0,1024);
```

**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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// scan to 50V  
result = SA_ScanMoveAbsolute_S(0,0,2048,10000);  
// scan to 25V within one second  
result = SA_ScanMoveRelative_S(0,0,-1024,1024);  
// scan to (automatically stop at) 0V  
result = SA_ScanMoveRelative_S(0,0,-3000,1024);
```

**See also:** `SA_ScanMoveAbsolute_S`, `SA_GetVoltageLevel_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. See also `SA_GotoAngleAbsolute_S`.

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 - Selects the system. The index is zero based.
- *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(0,0,0,0);
```

**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. See also `SA_GotoAngleRelative_S`.

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 - Selects the system. The index is zero based.
- *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(0,0,-2000,0);
```

**See also:** `SA_GotoPositionAbsolute_S`, `SA_GotoAngleRelative_S`, `SA_GetPosition_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. See also `SA_GotoPositionAbsolute_S`.

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 5.1 “Rotary Sensors” for more information on the angle and revolution parameters.

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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(0,0,90000000,0,1000);
```

**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. See also `SA_GotoPositionRelative_S`.

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 5.1 “Rotary Sensors” for more information on the angle and revolution parameters.

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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(0,0,-45000000,-1,0);
```

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

# 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 - Selects the system. The index is zero based.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// perform 1,000 steps with half amplitude at 1kHz  
result = SA_StepMove_S(0,0,1000,2048,1000);  
// stop  
result = SA_Stop_S(0,0);  
/*
```

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.

```
*/
```

# 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 in the range of about 100µm. The user 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`).

## Parameters:

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

## Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// start calibration routine  
result = SA_CalibrateSensor_S(0,0);  
unsigned int status;  
do {  
    SA_GetStatus_S(0,0,&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 a reference mark, this function may be used to move the positioner to the reference mark. It marks a known physical position of the positioner. The user must specify the search direction. If a mechanical end stop is detected before the reference mark is found, the positioner will inverse the search direction automatically. If the auto zero flag is set, the current position resp. angle is set to zero after the mark 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 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 - Selects the system. The index is zero based.
- *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_UP_DIRECTION` or `SA_DOWN_DIRECTION`.
- *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_SetZeroPosition_S`).

## Example:

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

# 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 farther 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 - Selects the system. The index is zero based.
- *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(0,0,0,10000);  
// wait until gripper is open  
do {  
    SA_GetStatus_S(0,0,&status);  
} while (status != SA_STOPPED_STATUS);  
// close gripper to 50V within two seconds  
SA_GotoGripperOpeningAbsolute_S(0,0,5000,25);
```

**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 - Selects the system. The index is zero based.
- *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(0,0,0,10000);  
// wait until gripper is open  
do {  
    SA_GetStatus_S(0,0,&status);  
} while (status != SA_STOPPED_STATUS);  
// close gripper by 25V within half a second  
SA_GotoGripperOpeningRelative_S(0,0,2500,50);
```

**See also:** `SA_GotoGripperOpeningAbsolute_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 - Selects the system. The index is zero based.
- *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  $\mu\text{N}$ , e.g. a value of 100 would be  $10\mu\text{N}$ . 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. 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:

```
// grab with 20 $\mu\text{N}$ , do not move the gripper faster than 100 V/s  
SA_GotoGripperForceAbsolute_S(0,0,200,100);
```

**See also:** `SA_SetZeroForce_S`

# 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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get current voltage level  
unsigned int level;  
result = SA_GetVoltageLevel_S(0,0,&level);  
if (result == SA_OK) {  
    // voltage level is in level variable  
}
```

**See also:** `SA_ScanMoveAbsolute_S`, `SA_ScanMoveRelative_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.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get current position  
signed int position;  
result = SA_GetPosition_S(0,0,&position);  
if (result == SA_OK) {  
    // current position is in position variable  
}
```

**See also:** `SA_GetAngle_S`

# 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 5.1 “Rotary Sensors” for more information on the angle and revolution parameters.

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get current position  
signed int angle, revolution;  
result = SA_GetAngle_S(0,0,&angle,&revolution);  
if (result == SA_OK) {  
    // angle and revolution parameters have been updated  
}
```

**See also:** `SA_GetPosition_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 the appendix 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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get current status  
unsigned int status;  
result = SA_GetStatus_S(0,0,&status);
```

## 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 - Selects the system. The index is zero based.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get current voltage  
unsigned int voltage;  
result = SA_GetGripperOpening_S(0,0,&voltage);
```

**See also:** `SA_GotoGripperOpeningAbsolute_S`, `SA_GotoGripperOpeningRelative_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 is 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 - Selects the system. The index is zero based.
- *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  $\mu\text{N}$ .

### Example:

```
SA_STATUS result = SA_InitSystems(SA_SYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// get current status  
unsigned int status;  
result = SA_GetForce_S(0,0,&status);
```

**See also:** `SA_SetZeroForce_S`

### 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\_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 the functions described below.

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.

**Parameters:**

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

**Example:**

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);
if (result != SA_OK) {
    // handle error...
}
// request current movement speed
result = SA_GetClosedLoopMoveSpeed_A(0,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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 5.3 “Defining Positions”.

## Parameters:

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

## Example:

```
// request known-status  
result = SA_GetPhysicalPositionKnown_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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

### Example:

```
// request range limit  
result = SA_GetPositionLimit_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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

### Example:

```
// request range limit  
result = SA_GetAngleLimit_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 “completed” packet when it has completed the movement. See section 4.5.1 “Report on Complete” for more information. The default behavior is no reporting.

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *report* (unsigned 32bit), input - Must be SA\_NO\_REPORT\_ON\_COMPLETE or SA\_REPORT\_ON\_COMPLETE.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
result = SA_SetReportOnComplete_A(0,0,SA_REPORT_ON_COMPLETE);  
  
// See also the code example of SA_SetReceiveNotification_A
```

## 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 the functions described below.

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 - Selects the system. The index is zero based.

### Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);
if (result != SA_OK) {
    // handle error...
}
// request current sensor mode
result = SA_GetSensorEnabled_A(0,0);
SA_PACKET packet;
// wait for answer, but not longer than one second
result = SA_ReceiveNextPacket_A(0,1000,&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 the functions described below.

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

## Parameters:

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

## Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current sensor type  
result = SA_GetSensorType_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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 - Selects the system. The index is zero based.
- `channelIndex` (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current end effector type  
result = SA_GetEndEffectorType_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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

### Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current voltage level  
result = SA_GetVoltageLevel_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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.

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 - Selects the system. The index is zero based.
- `channelIndex` (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current position  
result = SA_GetPosition_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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.

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 - Selects the system. The index is zero based.
- `channelIndex` (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.

## Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current angle  
result = SA_GetAngle_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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 the appendix for a list of status codes.

### Parameters:

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

### Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current status  
result = SA_GetStatus_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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

## Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current gripper position  
result = SA_GetGripperOpening_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_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 the functions described below.

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  $\mu$ N.

### Parameters:

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

### Example:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request current force  
result = SA_GetForce_A(0,0);  
SA_PACKET packet;  
// wait for answer, but not longer than one second  
result = SA_ReceiveNextPacket_A(0,1000,&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\_SetReceiveNotification\_A

## Interface:

```
SA_STATUS SA_SetReceiveNotification_A(SA_INDEX systemIndex,  
                                     HANDLE event);
```

## Description:

This function may be used to inform an application when a data packet has been received from the hardware. The application (or a thread of the application) may create an event, call this function and then block on the event. When a packet has been received, the application (thread) is unblocked and may issue the reception of the data. See section 4.3.2 "Event Driven Answer Retrieval" for more information.

## Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *event* (HANDLE), input - Specifies the event handle that is to be signaled on data reception.

## Example:

```
// Note: In this example the function results are not checked!  
  
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
// Create an event to block on until a data packet is received.  
HANDLE event = CreateEvent(  
    NULL,  
    false,          // auto-reset event  
    false,         // non-signaled state  
    NULL  
);  
result = SA_SetReceiveNotification_A(0,event);  
// Tell channel to report the completion of movement commands.  
result = SA_SetReportOnComplete_A(0,0,SA_REPORT_ON_COMPLETE);  
// Move positioner, 1,000 steps, full amplitude, 100 Hz.  
result = SA_StepMove_A(0,0,1000,4095,100);  
// Wait until channel reports completion. This function blocks  
// until data is received.  
WaitForSingleObject(event,INFINITE);  
// Receive packet.  
SA_PACKET packet;  
SA_INDEX channel;  
result = SA_ReceiveNextPacket_A(0,0,&packet);  
// Check if the answer is the one we're expecting.  
if (packet.packetType == SA_COMPLETED_PACKET_TYPE) {  
    if (packet.channelIndex == 0) {  
        // positioner completed movement.  
    } // else handle error...  
} // else handle error...
```

## 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 4.3.2 “Event Driven Answer Retrieval” for more information.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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 0 will let the function return immediately, whether data was received or not.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request position of positioner  
result = SA_GetPosition_A(0,0);  
// receive answer, but don't wait longer than five seconds  
SA_PACKET packet;  
result = SA_ReceiveNextPacket_A(0,5000,&packet);
```

**See also:** `SA_ReceiveNextPacketIfChannel_A`, `SA_LookAtNextPacket_A`, `SA_DiscardPacket_A`

## SA\_ReceiveNextPacketIfChannel\_A

### Interface:

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

### Description:

This function is similar to the previous one (see there). It 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 no packet was received then a packet of type `SA_NO_PACKET_TYPE` is returned. If a packet is received and originates from the specified channel it is returned and consumed by the call. If a packet is received that does not originate from the specified channel then a packet of type `SA_NO_PACKET_TYPE` is returned and the received packet is left in the receive buffer. See section 4.3.2 “Event Driven Answer Retrieval” for more information.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *channelIndex* (unsigned 32bit), input - Selects the channel of the selected system. The index is zero based.
- *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 0 will let the function return immediately, whether data was received or not.
- *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:

```
SA_STATUS result = SA_InitSystems(SA_ASYNCHRONOUS_COMMUNICATION);  
if (result != SA_OK) {  
    // handle error...  
}  
// request position of positioner  
result = SA_GetPosition_A(0,0);  
// receive answer, but don't wait longer than five seconds  
SA_PACKET packet;  
result = SA_ReceiveNextPacketIfChannel_A(0,0,5000,&packet);
```

**See also:** `SA_ReceiveNextPacket_A`, `SA_LookAtNextPacket_A`, `SA_DiscardPacket_A`

## 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 4.3.2 “Event Driven Answer Retrieval” for more information.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.
- *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 0 will let the function return immediately, whether data was received or not.
- *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(0,0);
// wait until answer is there
SA_PACKET packet;
SA_LookAtNextPacket_A(0,1000,&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_ReceiveNextPacketIfChannel_A`, `SA_DiscardPacket_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 4.3.2 “Event Driven Answer Retrieval” for more information.

### Parameters:

- *systemIndex* (unsigned 32bit), input - Selects the system. The index is zero based.

### Example:

```
// request position of positioner
result = SA_GetPosition_A(0,0);
// 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(0,1000,&packet);
    if (packet.packetType != SA_POSITION_PACKET_TYPE)
        SA_DiscardPacket_A(0);
}
```

**See also:** SA\_ReceiveNextPacket\_A, SA\_ReceiveNextPacketIfChannel\_A, SA\_LookAtNextPacket\_A

## 4 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.

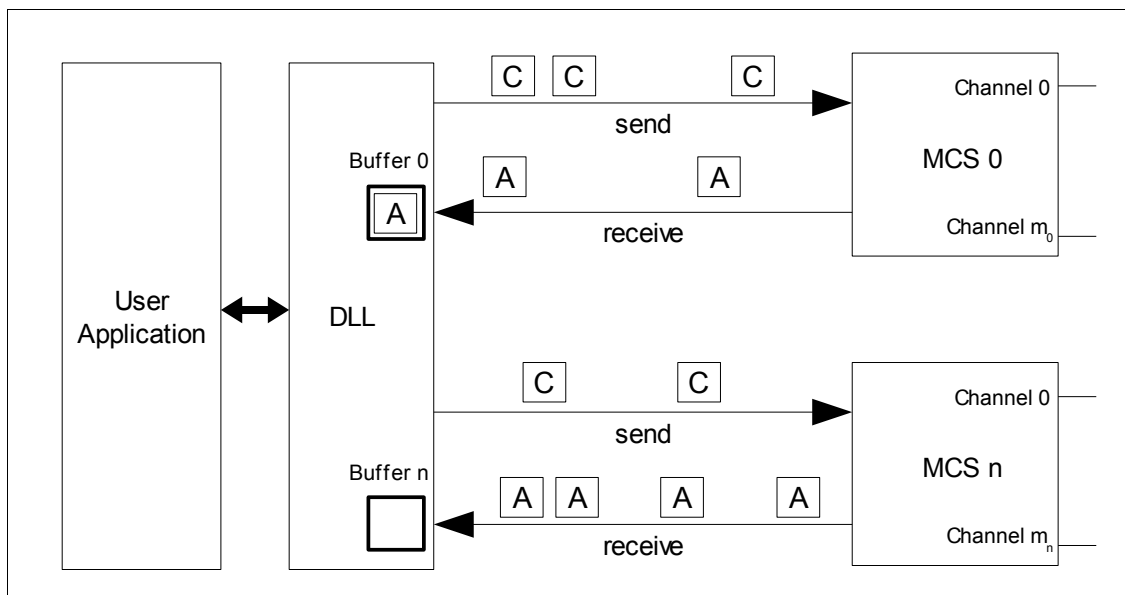
### 4.1 Overview

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. Also, if several commands are sent to different channels of a system, the order of answers (if any) is not defined.

Therefore, 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 DLL supports communicating with several systems in parallel.



### 4.2 Sending Commands

Most functions of section IIb of the DLL 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 DLL 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.

## 4.3 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 DLL offers several ways of handling answer retrieval.

Generally, answers coming from an MCS are put into a receive buffer (see figure above). The DLL 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.

Several functions are used to access the receive buffer:

- `SA_ReceiveNextPacket_A`:  
This is the standard answer retrieval function. It returns the answer packet that is currently located in 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 in 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.
- `SA_ReceiveNextPacketIfChannel_A`:  
This function behaves the same way as `SA_ReceiveNextPacket_A`, but only returns and consumes a packet if it was sent by a specific channel of the system. If a packet was received that originates from another channel, then a packet of type `SA_NO_PACKET_TYPE` is returned and the received packet remains in the buffer.

Some implications:

- “Looking at” a packet stops the data flow, since the packet is not consumed. In order to receive further packets, call `SA_ReceiveNextPacket_A` or `SA_DiscardPacket_A`.
- In a similar manner, using `SA_ReceiveNextPacketIfChannel_A` conditionally stops the data flow.
- 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).

### 4.3.1 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 the appendix for the list of packet types and their meanings.

### 4.3.2 Event Driven Answer Retrieval

Often, the polling technique to receive data packets from the hardware is undesired since it uses CPU time unnecessarily. Therefore, the DLL offers a mechanism to notify the user application when a data packet has been received.

The user application can create an event object (`CreateEvent`, defined in `windows.h`) and pass it to the `SA_SetReceiveNotification_A` function. The event object gets signaled whenever a data packet has been received. Thus, the application (thread) can block on the event object (`WaitForSingleObject`) and will be woken up when there is data to be processed. See the `SA_SetReceiveNotification_A` function for a simple code example.

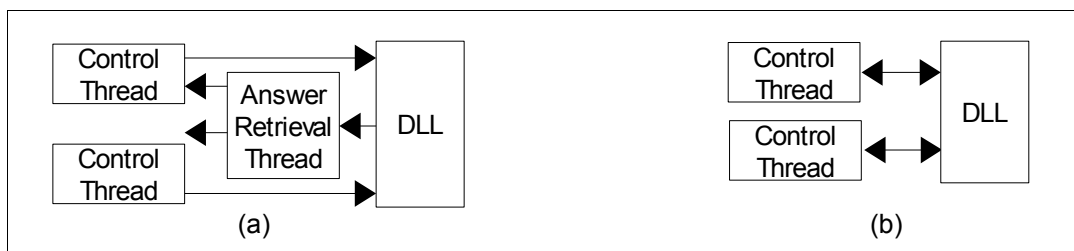
Note: When using the event notification mechanism, there is no need to set a timeout when calling the functions `SA_ReceiveNextPacket_A`, `SA_LookAtNextPacket_A` or `SA_ReceiveNextPacketIfChannel_A`.

An event object may have the state “signaled” (in this context: “a packet has been received”) or “non-signaled” (“no packet has been received”). When creating the event with `CreateEvent` the user has the choice of creating an auto reset event or a manual reset event. The setting and resetting of the state of the event object is completely handled by the DLL. Therefore, it generally doesn't matter whether using an auto reset event or a manual reset event. Note that when using auto reset events in multi threaded applications (see next section), if an unblocked thread only “looks at” a packet without consuming it, the DLL will set the event object to the signaled state again and will continue to unblock threads until the packet is consumed. Generally, the state of the event object will always be “signaled” if there is a packet (remaining) in the receive buffer.

There are, however, certain situations where the user might want to set the event object to the signaled state manually. Please refer to section 4.4.1 “Thread Termination” on this topic.

### 4.4 Multi Threaded Applications

While it is possible to implement single threaded applications using the asynchronous mode, its main intent are multi threaded applications utilizing the event driven answer retrieval mechanism. All functions of the DLL are thread safe. There are several possibilities of architectural application design using multiple threads. Two of these shall be shown here as an example.



In example (a) there are several control threads that might each control one channel of the system. They send commands by calling DLL functions directly and receive answers through a further thread that is dedicated for answer retrieval. This thread forwards answer packets to the appropriate control threads depending on which one is responsible for each packet. As an example, the implementation of the answer retrieval thread would look something like this:

```
// create event for receive notification
HANDLE packetReceived = CreateEvent(NULL, false, false, NULL);
SA_SetReceiveNotification_A(0, packetReceived);
SA_PACKET packet;
// keep receiving until told to terminate
WaitForSingleObject(packetReceived, INFINITE);
while (!terminate) {
    SA_ReceiveNextPacket_A(0, 0, &packet);
    forwardPacket(packet.channelIndex, packet);
    WaitForSingleObject(packetReceived, INFINITE);
}
```

Please note that in this simple example neither the return status code of the function calls nor the packet types are checked. (For example, should the packet type be `SA_SENSOR_ENABLED_PACKET_TYPE`, the `channelIndex` field is not defined!)

In example (b) there are also several control threads, but they each retrieve their answer packets through the DLL directly. All threads block on the same globally created event and “look at” the next packet received, only consuming it if it originates from the desired channel. A sample code for a control thread querying the current position of a positioner would look something like this:

```
// request position of channel myChannel.
SA_GetPosition_A(0,myChannel);
SA_PACKET packet;
// wait until channel answers.
do {
    // (packetReceived object was created in main application thread.)
    WaitForSingleObject(packetReceived,INFINITE);
    // take a look at the received packet without consuming it.
    SA_LookAtNextPacket_A(0,0,&packet);
} while (packet.channelIndex != myChannel);
// received a packet for channel myChannel. consume packet.
SA_DiscardPacket_A(0);
if (packet.packetType == SA_POSITION_PACKET_TYPE) {
    // position is in packet.data2.
}
}
```

Note: In this example it must be assured that all received packets are consumed by a thread. Otherwise the data flow will stop and the threads will loop forever leading to an overall deadlock.

#### 4.4.1 Thread Termination

When the application is up and running there is typically at least one thread blocking on the receive event object waiting for incoming data packets. When the application is to be terminated (potentially on user request) all blocked threads need to be unblocked for a proper cleanup. For this, the event object may be set to the signaled state manually by calling `SetEvent` (defined in `windows.h`). If more than one thread is blocking on the event object, it becomes relevant if the object was created as an auto reset or a manual reset event. With a manual reset event all blocking threads will be unblocked and they can take appropriate steps to terminate themselves. With an auto reset event only one thread is unblocked. In this case the main application thread should call `SetEvent` as often as there are blocked threads. Alternatively, the first unblocked thread can call `SetEvent` before terminating, thereby unblocking the next thread.

### 4.5 Other Issues

#### 4.5.1 Report on Complete

The DLL 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 the appendix 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 1: 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 2: 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).

## 4.5.2 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 altogether. See the SA\_SetHCMEnabled function.

## 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.

### 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_GotoAngleAbsolute_X`, 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_X` 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(0,0,-90000000,0,0);  
SA_GotoAngleRelative_S(0,0,270000000,-1,0);
```

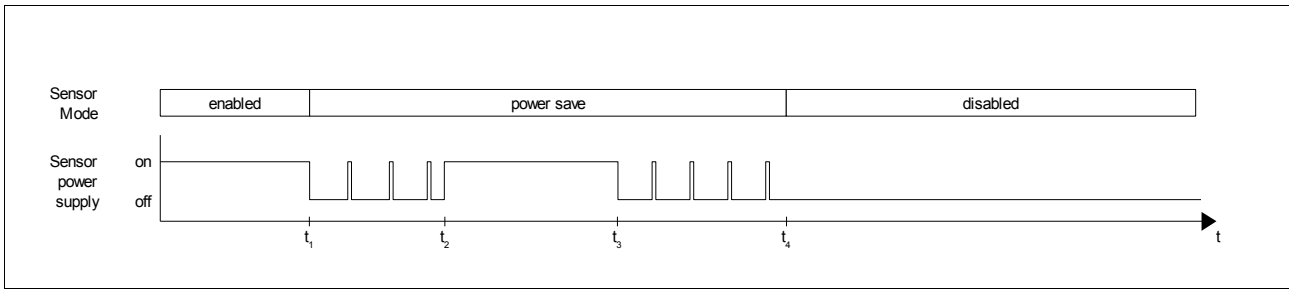
Both commands will move the positioner 90° in negative direction.

### 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_X` 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.
- **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.

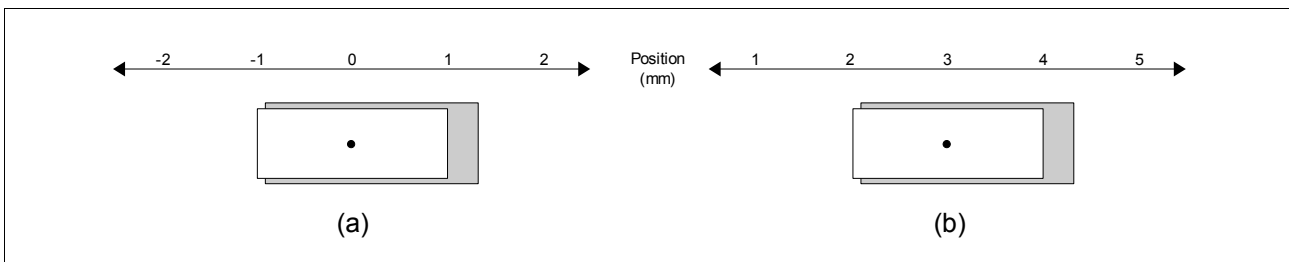
**Note on the power save mode:** If closed-loop commands are issued with a hold time, then the system will start to pulse the power supply of the sensor as soon as the target position has been reached. At this point the hold time starts. The positioner will still hold the target position and compensate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

### 5.3 Defining Positions

Since the position calculation is done on an incremental basis, a positioner has no way of knowing its physical position 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_X` 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, which makes it 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(0,0,3000000)`; The current position has been defined to +3mm and the measuring scale is shifted accordingly.



#### 5.3.1 Reference Marks

In the example above the physical position of a positioner has to 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 SmarAct's positioners are equipped with reference marks. These marks are detectable by the system when moving the positioner over its travel range and enables the system to determine the physical position in an automated fashion. The `SA_FindReferenceMark_X` function is used for this purpose. Depending on the product option of your positioner it may be equipped with a single reference mark or with multiple reference marks. The search algorithms are outlined in the following:

- **Single Reference Mark:** In this case the reference mark 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. Should the positioner detect 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.)

- 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 then the current position value is stored and the search continues in the same direction for the second mark. 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.)

When the command has completed successfully the system knows the physical position of the positioner.

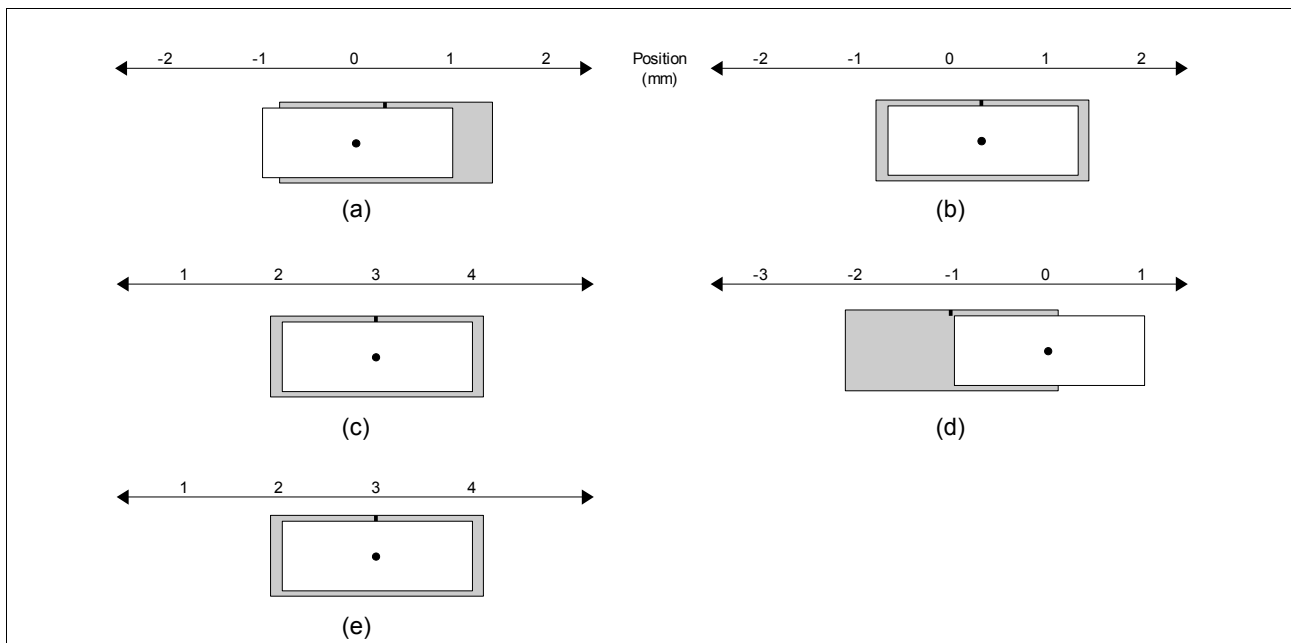
### 5.3.2 Recalling Positions

Using the reference marks to determine the physical position of a positioner it is possible to recall a user defined position after a system power-up.

If an arbitrary position is defined (`SA_SetPosition_X`) *after* the physical position has been determined (`SA_FindReferenceMark_X`) then this user defined position is automatically stored to non-volatile memory. On any future power-ups the position will automatically be recalled after a `SA_FindReferenceMark_S`.

The figure below shows an example. The code section shows the corresponding command sequence.

```
// (a) - System is powered up. The current position is assumed to be 0.
//       The small marking indicates the position of the reference mark.
SA_FindReferenceMark_S(0,0,SA_FORWARD_DIRECTION,0,SA_NO_AUTO_ZERO);
// (b) - The positioner has moved to the reference mark and its current
//       position is slightly positive.
SA_SetPosition_S(0,0,3000000);
// (c) - The measuring scale has been shifted. Since the reference mark was
//       found before, the position is saved to non-volatile memory.
// (d) - The system was shut down, the positioner was moved externally to
//       some random position and the system is then powered up again.
//       The positioner assumes its current position as 0 again.
SA_FindReferenceMark_S(0,0,SA_BACKWARD_DIRECTION,0,SA_NO_AUTO_ZERO);
// (e) - The positioner has moved to the reference mark and the position
//       that was saved earlier has been recalled automatically, i.e.
//       there is no need to call SA_SetPosition_S again.
```

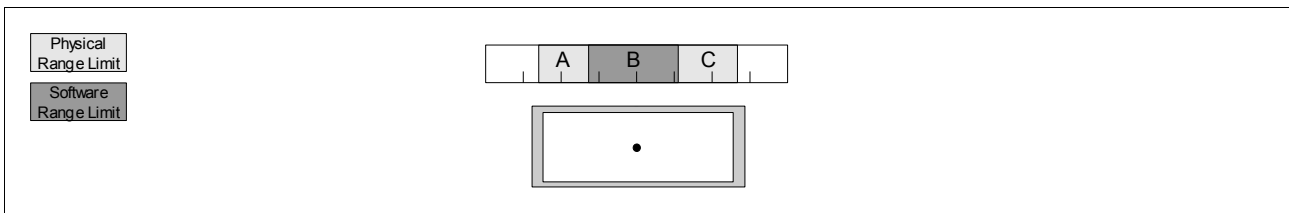


## 5.4 Software Range Limit

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.

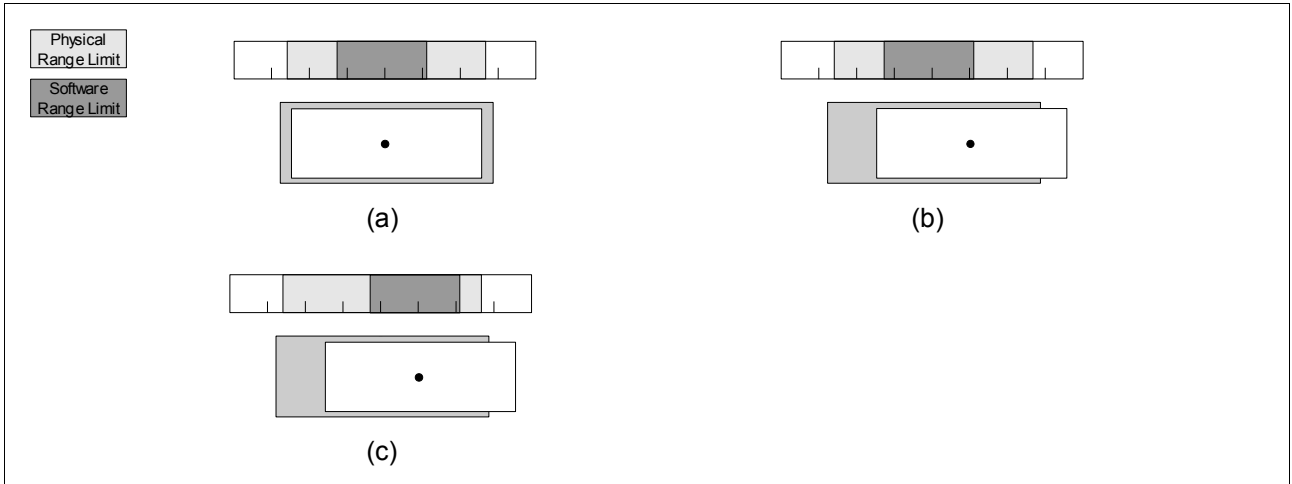


Please note the following restrictions:

- 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. This means that shifting the measurement scale of the positioner with this command 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.



## 6 Appendix

### 6.1 Status / Error Codes

| Code | Symbol<br>Description   |
|------|---|
| 0    | SA_OK<br>The function call was successful.  |
| 1    | SA_INITIALIZATION_ERROR<br>An error occurred while initializing the DLL. All systems should be disconnected and reset before the next attempt is made.  |
| 2    | SA_NOT_INITIALIZED_ERROR<br>A function call has been made while the DLL not being initialized. You must call <code>SA_InitSystems</code> before communicating with the hardware.  |
| 3    | SA_NO_SYSTEMS_FOUND_ERROR<br>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<br>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<br>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<br>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<br>An error occurred while sending data to the hardware. The system should be reset.  |
| 8    | SA_WRITE_ERROR<br>An error occurred while sending data to the hardware. The system should be reset.   |
| 9    | SA_INVALID_PARAMETER_ERROR<br>An invalid parameter has been passed to a function. Check the function documentation for valid ranges.  |
| 10   | SA_READ_ERROR<br>An error occurred while receiving data from the hardware. The system should be reset.  |
| 12   | SA_INTERNAL_ERROR<br>An internal communication error occurred. The system should be reset.  |
| 13   | SA_WRONG_MODE_ERROR<br>The called function does not match the communication mode that was selected at initialization (see <code>SA_InitSystems</code> ). 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<br>An internal protocol error occurred. The system should be reset.   |
| 15   | SA_TIMEOUT_ERROR<br>The hardware did not respond. Make sure that all cables are connected properly and reset the system.  |

- 16 SA\_NOTIFICATION\_ALREADY\_SET\_ERROR  
Since all channels of a system share the same receive buffer, the receive notification object can only be set once per system. Multiple calls of SA\_SetReceiveNotification\_A lead to this status code.
- 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.
- 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. If an end stop is detected during the search, the search direction is reversed. If a second end stop is detected, the search is aborted.
- 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.
- 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.
- 240 `SA_UNKNOWN_COMMAND_ERROR`  
This error occurs if the DLL sends a command that is not supported by the system it is sent to. This may be the case when using a newer DLL with an older firmware version. Please be sure only to use the DLL that is shipped with your system to avoid compatibility problems.
- 255 `SA_OTHER_ERROR`  
An error that can't be otherwise categorized.

## 6.2 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<br>Description   | Valid fields                      |
|------|---|-----------------------------------|
| 0    | SA_NO_PACKET_TYPE<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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               |



## 6.3 Channel Status Codes

| 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 a stepping movement.   |
| 2    | SA_SCANNING_STATUS    | The positioner is currently performing a scanning movement.   |
| 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.  |