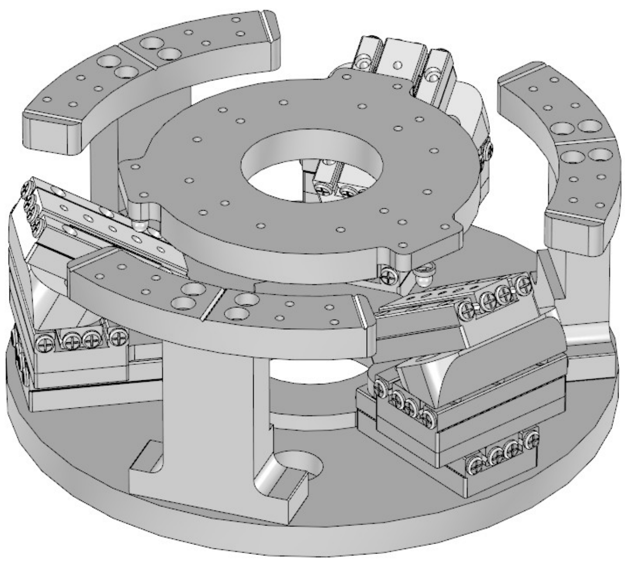
# SmarPod
# Programmer's Guide



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

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

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

Document Version 14-08-05/1771
Library Version 1.5.16

# Table of Contents

# 1 Introduction

The *SmarPod Programmer's Guide* describes the *C* application programming interface (API) for *SmarAct SmarPod* manipulators.

Chapter 2 gives an overview over the general concepts and terminology of the SmarPod.

Chapter 3 contains tips for troubleshooting.

Chapters 4 and 5 document the API data types and functions.

Chapter 6 (Appendix) has programming examples, a table of properties and lists the API status codes.

## 1.1 Changes in SmarPod API Version 1.5

In the API version 1.5 new functions have been added: `Smarpod_Open`, `Smarpod_FindSystems`, `Smarpod_GetSystemLocator` and `Smarpod_Close`. The older functions `Smarpod_InitSystems`, `Smarpod_Initialize` and `Smarpod_ReleaseAll` are still in the API but are **deprecated** because the new functions are much more flexible and easier to handle. Only with `Smarpod_Open` it is possible to connect to a MCS controller with a network interface.

We recommend not to use the older functions anymore.

# 2 Getting Started

## 2.1 Technical Overview

A SmarPod has three groups of positioners: *A*, *B* and *C*. Each group consists of one positioner oriented radially to the center, called *A/radial*, *B/radial* and *C/radial* and one positioner perpendicular to the radial one, called *A/tangential*, *B/tangential* and *C/tangential* respectively. The following illustrations show the layout and the coordinate system. Note the positions of the mounting holes and the cable.

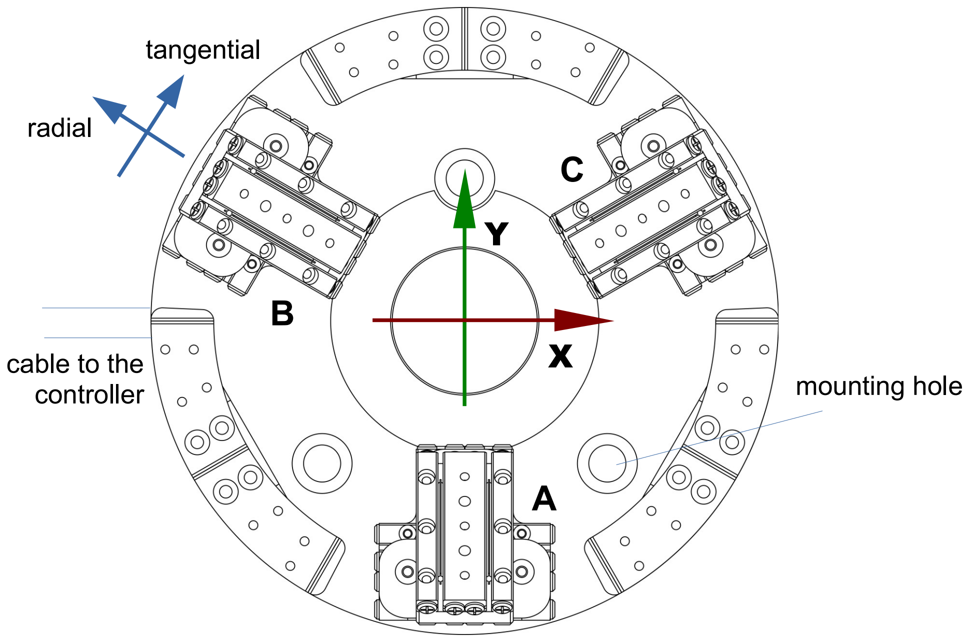


*Figure 1: A rotation-symmetric SmarPod (without the stage) from above*

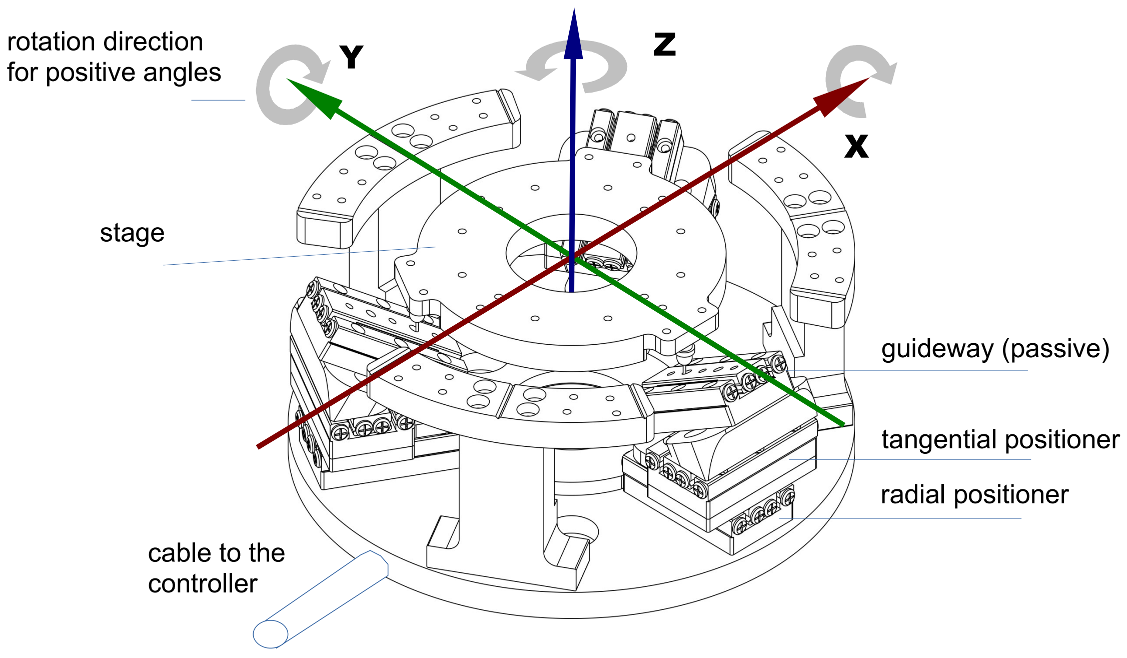


*Figure 2: The coordinate system axes of rotation-symmetric SmarPods*

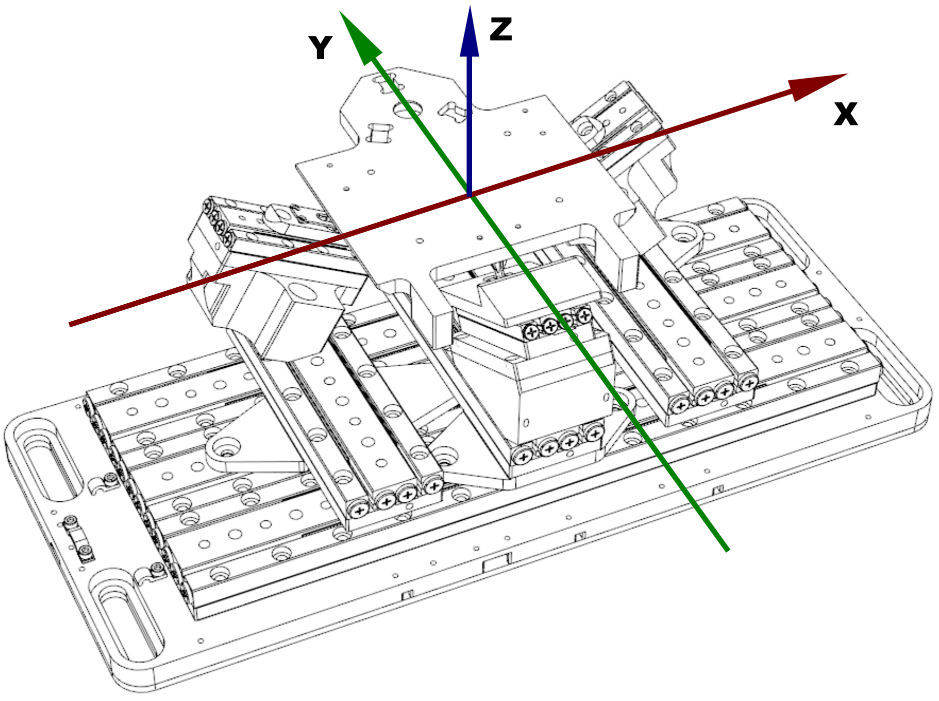


*Figure 3: The coordinate system axes of parallel SmarPods with 6 positioners*

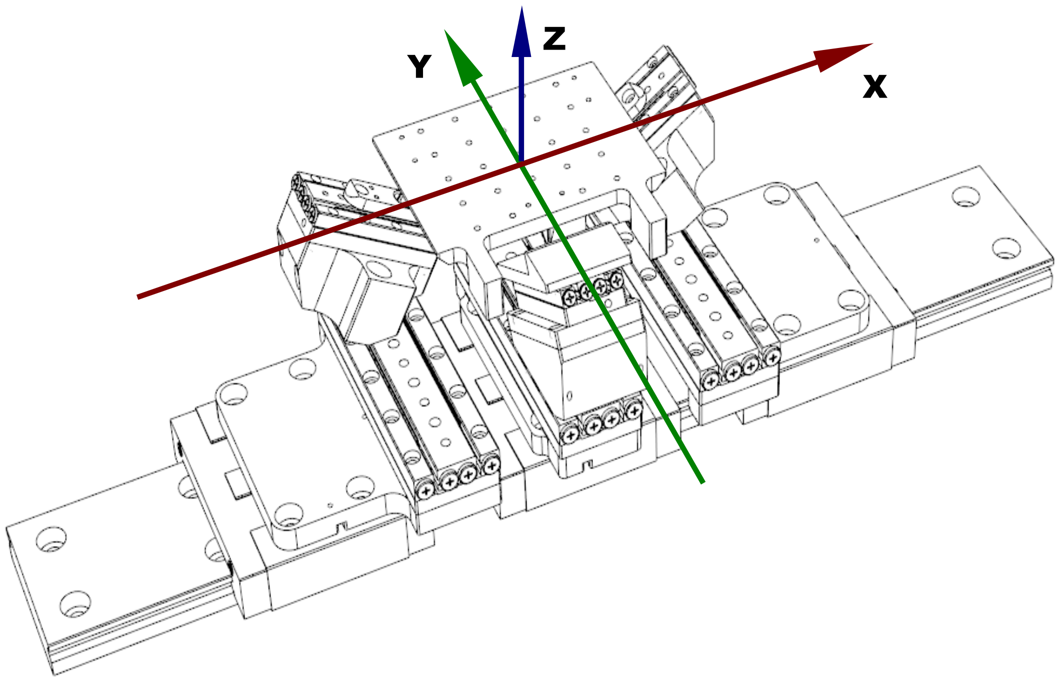


*Figure 4: The coordinate system axes of parallel SmarPods with a single rail in X direction*

## 2.2 Sensor Modes

In order to track its position, a sensor needs to be supplied with power. However, since this generates heat which may cause drift, it might be desirable to disable the sensors in some situations. There are three different modes of operation for the sensor, which may be configured with the `Smarpod_SetSensorMode` function.

- *Disabled* (parameter `SMARPOD_SENSORS_DISABLED`) – In this mode the power supply of the sensor is turned off. Commands such as `Smarpod_Move` will fail with an error. This mode may also be useful if the light that is emitted by the sensors interferes with other components of your setup.
- *Enabled* (parameter `SMARPOD_SENSORS_ENABLED`) – In this mode the sensor is supplied with power continuously. All movement commands are executed normally.
- *Power Save* (parameter `SMARPOD_SENSORS_POWERSAVE`) – 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 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.

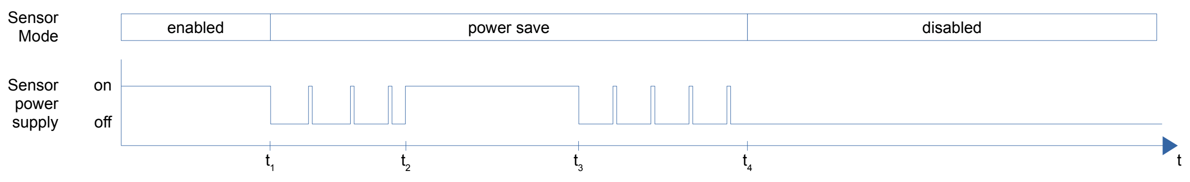


*Figure 5: Sensor Modes and Power*

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 the system will not be able to track the position. The position data will become invalid.

**Notes on the power save mode:**

If movement commands are issued with a hold time, 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.

When a movement command is issued in power save mode the sensors have to be temporarily enabled before the movement can begin. Powering up the sensors takes a few milliseconds.

## 2.3 Properties

The SmarPod API provides functions to write and read *properties* which configure the behavior of the SmarPod and the SmarPod software. Properties are written with the `Smarpod_Set_...` and read with the `Smarpod_Get_...` functions. A list of properties and values is listed in appendix 6.2 “SmarPod Properties And Values”.

Not all properties can be set to every value under all circumstances. If an unknown property is passed to `Smarpod_Set_...` or `Smarpod_Get_...` `SMARPOD_UNKNOWN_PROPERTY_ERROR` error is returned. If an invalid value is passed, `SMARPOD_INVALID_PARAMETER_ERROR` is returned.

## 2.4 Initialization

A program must initialize and configure a SmarPod before it can be used. The steps are as follows:

1. initialization of the SmarPod (see 2.4.1 “Initializing a SmarPod“).
2. activation of the sensors operation mode (see 2.2 “Sensor Modes“).
3. configuration of the SmarPod (2.4.3 “Configuring the SmarPod”).
4. (if necessary) calibration of the positioner sensors (see 2.4.4 “Calibrating the Sensors“).
5. (if necessary) finding the positioner's reference marks (see 2.4.5 “Finding Reference Marks“).

### 2.4.1 Initializing a SmarPod

`Smarpod_Open` is used to initialize a SmarPod[1]. The MCS that controls the SmarPod must be described with a *device locator* string, which is similar to URLs in a web browser. Example:

```
unsigned int smarpodId;
Smarpod_Open(&smarpodId,10001,"usb:id:123456789","");
```

For an explanation of the locator syntax see 2.4.2 “Device Locators”. If the initialization is successful, `Smarpod_Open` returns a handle (*smarpodId*) which must be passed to the API functions.

**SmarPod Hardware Model**

The hardware model parameter of `Smarpod_Open` identifies the type of the SmarPod that is connected to the controller. The function `Smarpod_GetModels`. returns a list of all SmarPod model codes supported by the installed SmarPod software.

Please ensure that the right model code is used when initializing a SmarPod. Otherwise the initialization might succeed but the execution of commands can lead to undefined behavior.

### 2.4.2 Device Locators

**USB Device Locator Syntax**

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

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

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

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

where the number *<n>* selects the *n*th device in the list of all currently connected MCS with a USB interface. The drawback of identifying an MCS with this method is, that the number and order of the connected MCS can change between sessions, so the index *n* may not always refer to the same device. It is only safe to do this if you have exactly one MCS connected to the PC. We recommend to use the `usb:id:...` format for USB systems.

**Network Device Locator Syntax**

The network locator format is:

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

*<ip>* is an IP address which consists of four integer numbers between 0 and 255 separated by a dot. *<port>* is an integer number.

Do not add leading zeros to the numbers in the IP address as these would be interpreted as *octal* numbers. The IP addresses 192.168.001.200 and 192.168.1.200 are **not** the same!

1: In API versions before 1.5 the functions `Smarpod_InitSystems` and `Smarpod_Initialize` were used for initialization. Since these are deprecated they are no longer explained in this chapter.

For example, the locator `network:192.168.1.200:5000` addresses a device with the IP address 192.168.1.200 and TCP port 5000.

Some points should be remembered when writing or extending software for network communication:

- MCS systems with network interface can only be initialized with the function `Smarpod_Open`.
- The location (IP and port) of a MCS with network interface must be known to the program.

### 2.4.3 Configuring the SmarPod

After initialization the internal parameters of the SmarPod are set to default values. The following configuration parameters should be set by the program before actually moving the SmarPod:

- The sensor power mode (with `Smarpod_SetSensorMode`),
- the speed (with `Smarpod_SetSpeed`) and
- the maximum frequency (with `Smarpod_SetMaxFrequency`).

### 2.4.4 Calibrating the Sensors

When a SmarPod is connected to a controller for the first time the sensors of its positioners must be calibrated. Every SmarPod is calibrated by before shipment so it is usually not necessary to do a calibration for a new SmarPod.

Calibration must not be started when the positioners are close to their mechanical end stops.

`Smarpod_Calibrate` calibrates one positioner at a time. When a positioner is calibrated it is moved in small steps forward and backward for a few seconds. This is all a nano-sensor positioner does when it is calibrated. It does not move greater distances during calibration.

Micro-sensor calibration works differently. After the small calibration movements the positioner moves to one of its mechanical end-stops and then back to the center position. Because a micro-sensor positioner needs full movement range during calibration, it is important to do a calibration of a micro-sensor SmarPod only when all the positioners are close to their center positions.

### 2.4.5 Finding Reference Marks

When a MCS is powered up it does not know the physical positions of the positioners. Before the SmarPod is ready for operation, the controller must move every positioner to a *reference mark* with a known physical position. This is done by the function `Smarpod_FindReferenceMarks` (*FRM* in this section). It is not necessary to call FRM every time a program is run. Only if `Smarpod_IsReferenced` returns *false* it must be called. Trying to move an unreferenced SmarPod results in a `SMARPOD_NOT_REFERENCED_ERROR`.

During the execution of FRM the positioners are moved with a fixed frequency, not with speed-control (see section 2.5.2 “Velocity“). The frequency that is used by FRM can be set with `Smarpod_SetMaxFrequency`.

#### Positioner Types

Depending on the type of the positioners used in a SmarPod, the reference marks differ in number and location and in the way they are handled by FRM.

| **Positioner Types** | |
|---|---|
| Single Reference Mark Nano-Sensors<br>e.g.<br>*SmarPod 110.45 S* | These positioners have one reference mark near the middle of their range.<br>If a positioner is at a position left of the reference mark and FRM moves it to the right, it will move directly to the reference mark and stops.<br>If it starts at a position right of the mark and FRM moves it to the right, it moves until it reaches the right physical end-stop, reverses the direction and moves back until it finds the reference mark. |
| Micro-Sensors<br>e.g. | This positioner type uses one of the physical end-stops as the reference mark.<br>A positioner is moved in the direction of the end-stop which is used as the reference mark |

| | |
|---|---|
| *SmarPod 110.45 M* | until it reaches the end-stop. |
| Distance Coded Reference Marks e.g. *SmarPod 110.45 SC* | The positioners have multiple reference marks distributed over their range at distances of a few millimeters between two marks. A positioner moves until it reaches the next reference mark. If it starts between the last reference mark and the physical end-stop, it moves to the end-stop and reverses the direction. When the first reference mark has been found it moves to the next mark and stops. |

Depending on the sensor type and the initial position of the positioners, the movements of FRM vary. If a single reference mark SmarPod like the *110.45 S* starts from a pose where all its positioners except one are exactly on their reference marks and one is between the reference mark and an end-stop, only one positioner will move, which results in a rotation of the stage.

FRM moves SmarPods without distance-coded sensors to zero. SmarPods with distance-coded sensors remain in the pose where reference-marks have been found. To get the pose call `Smarpod_GetPose`.

### Configuring FindReferenceMarks

FRM can be configured by setting properties with the `Smarpod_Set_...` functions. The property `SMARPOD_FREF_METHOD` selects the algorithm used by FRM. The methods currently available are *Sequential*, *Z-Safe* and *XY-Safe*, which can be further configured with other properties. Not all SmarPod models provide the same FRM options. For example, the positioners of a *SmarPod 110.45 M* always search for the reference mark in the same direction, therefore FRM always uses the *Sequential* method and the direction properties have no influence. `Smarpod_Set_...` returns with status `SMARPOD_FEATURE_UNAVAILABLE_ERROR` or `SMARPOD_PARAMETER_INVALID_ERROR` if it is called with a method or direction which is not supported by the SmarPod model.

**Referencing Properties**

| Method | Directions | |
|---|---|---|
| *DEFAULT* | The software selects the method that is usually the best for the SmarPod model. | |
| *METHOD_SEQUENTIAL* | References all positioners one after the other. Available only for rotation-symmetric SmarPods. | |
| *METHOD_ZSAFE* | Optimizes movements to prevent collisions in the Z direction and moves more than one positioner simultaneously. Available for some rotation-symmetric SmarPod models. | |
| | FREF_ZDIRECTION | If the *Z Safe* method is selected, the `FREF_ZDIRECTION` property can be used to specify the Z direction in which the SmarPod searches for the positioners reference marks. Can be set to `NEGATIVE` or `POSITIVE` and can be combined with the `REVERSE` flag if the SmarPod model uses distance-coded sensors (see below). If the direction is set to `NEGATIVE` the SmarPod stage moves down (`POSITIVE`: up) until all positioners reference marks are found. If a positioner reaches the physical end its range it moves backwards until until the reference mark is found. |
| *METHOD_XYSAFE* | This method allows to specify a safe direction for X and Y. It is only available for parallel SmarPods. **Parallel SmarPods without distance coded sensors**: If `FREF_XDIRECTION` is set to `POSITIVE`, the SmarPod first moves in the positive X direction until the positioners have all reached the end of their movement ranges. Then it moves back to the reference marks. Likewise for `FREF_YDIRECTION`. **Parallel SmarPods with distance coded sensors**: The positioners move in the specified direction until two reference marks have been found. If a positioner starts between the last reference mark and an end-stop, it reverses the direction and move a short distance in the opposite direction. | |
| | FREF_XDIRECTION | The safe *X* movement direction. Can be set to `NEGATIVE` or `POSITIVE` and combined with the `REVERSE` flag if the SmarPod model uses distance-coded sensors (see below). |

| | | |
|---|---|---|
| | FREF_YDIRECTION | The safe *Y* movement direction. Can be set to `NEGATIVE` or `POSITIVE` and combined with the `REVERSE` flag if the SmarPod model uses distance-coded sensors (see below). |
| ***Other Properties*** | | |
| *FREF_AND_CAL_FREQUENCY* | This property sets the frequency that is used when referencing (and calibrating) a positioner. If it is not set, the frequency set with `Smarpod_SetMaxFrequency` is used instead (see note below). | |

For some SmarPod models with distance coded reference marks, the direction value can be combined with the *reverse* flag by *or*-ing the constants together, for example (in *C* syntax):

```
direction = SMARPOD_NEGATIVE | SMARPOD_REVERSE;
```

If the flag is set, FRM moves the positioners to the next mark in the specified direction and then in the opposite direction to find the second mark which minimizes the movement range of the SmarPod during FRM.

The *Z-Safe* method works best with distance coded reference marks. While it can also be used with single reference mark sensors, in this case it should only be used when the positioners are close to and move towards their reference marks when FRM is called. For example, if a nano-sensor SmarPod is moved to the pose Z = +1mm with a Z-rotation of -1° before the MCS is switched off:

```
Smarpod_Pose parkingPose = { 0.,0.,1e-3,0.,0.,-1. };
Smarpod_Move(0,&parkingPose,1);
```

and FRM is configured to move *Z-Safe* in negative Z direction:

```
Smarpod_Set_ui(0,SMARPOD_FREF_METHOD,SMARPOD_METHOD_ZSAFE);
Smarpod_Set_ui(0,SMARPOD_FREF_ZDIRECTION,SMARPOD_NEGATIVE);
Smarpod_Set_ui(0,SMARPOD_FREF_AND_CAL_FREQUENCY,8000);
Smarpod_FindReferenceMarks(0);
```

the resulting movements are short and have small rotation angles.

**Note**: in API versions before 1.5.11 `Smarpod_SetMaxFrequency` or the property `FREF_AND_CAL_FREQUENCY` could be used to set the frequency for referencing and calibration. For backward compatibility `Smarpod_SetMaxFrequency` can still be used to set the referencing and calibration frequency but has been **deprecated**.

### Configuring Speed-Controlled Movements

Some SmarPod models must perform additional speed-controlled movements when referencing to move the positioners to defined positions before starting the search for the reference marks. This is generally true for all parallel SmarPods without distance coded reference marks.

The speed parameters can be set with `Smarpod_SetSpeed` and `Smarpod_SetMaxFrequency`. The referencing frequency can be set independently from the max. frequency with the `SMARPOD_FREF_AND_CAL_FREQUENCY` property. (For backward compatibility, the frequency set with `Smarpod_SetMaxFrequency` will be used if this property is not set).

### Avoiding Collisions

There are several ways to reduce the risk of collisions when FRM is running:

- Make sure the movable parts of the SmarPod and objects mounted on it have enough space to move over the full range without colliding with other objects.
- Before switching off the SmarPod move it to a pose from which FRM can find the reference marks quickly without moving the positioners over long distances (see above).
- Configure the FRM function (see above).
- Use SmarPods with distance coded reference marks which have been designed to minimize movements when referencing.

**Overload Problems**

Depending on the SmarPod model and its configuration, FRM can fail because the power required when moving the positioners overloads the controller or the power-supply. In this case FRM should be configured to minimize power consumption, for example by using a FRM method that references fewer positioners at a time or by reducing the frequency and/or the speed.

Please read the notes regarding failures of movement commands caused by overload in section 3.2 “Movement Failures Caused By Overloading“.

## 2.5 *Moving the SmarPod*

The stage of the SmarPod can be moved in three directions and rotated around three axes. See section 2.1 “Technical Overview” for illustrations of the coordinate system.

### 2.5.1 Pose and Pivot Point

The translation and orientation of the SmarPod stage is called the *pose*. The translation part of a pose is the offset of the stage in *X*, *Y* and *Z* direction relative to the zero position. The angles $\Theta_x$, $\Theta_y$ and $\Theta_z$ define the rotation around the respective axis. The actual rotations depend on the *pivot point* which is the center of rotations. By setting the pivot point it is possible to let the SmarPod rotate around objects or points of interest. Since the pivot point is changed not very often it is not a parameter of the move command. Use `Smarpod_SetPivot` to change it. After initialization the pivot point is (0,0,0). With a pivot of (0,0,0) the stage rotates around the centroid of the triangle of the joints between the stage and the passive guideways (see Figure 6).

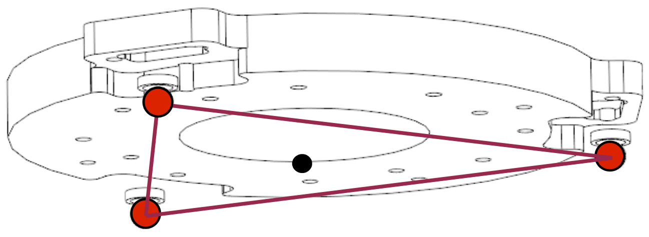


*Figure 6: The default pivot point*

The default pivot is below the stage because at this point the range of rotation angles is maximal. The greater the distance of the pivot to the default point, the smaller the angles the SmarPod can rotate.

Figure 7 illustrates how the stage is rotated by the same angles Θ and -Θ around different pivot points.

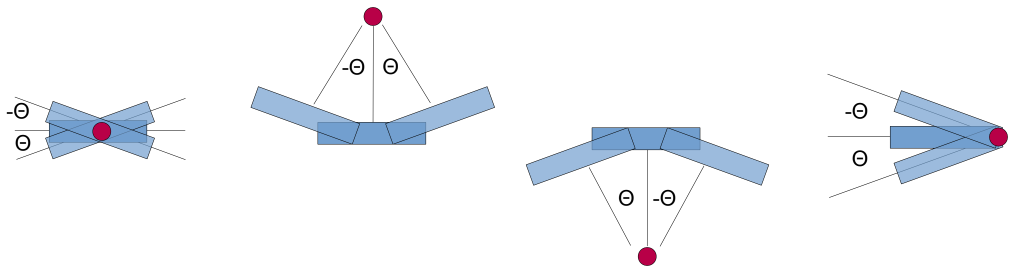


*Figure 7: Rotations of the stage around different pivot points*

The pivot point can be configured to be *relative* to the stage or *fixed*. This is controlled by the `SMARPOD_PIVOT_MODE` property. See 6.2 “SmarPod Properties And Values“. If the pivot point is relative to the stage, it follows the stage translation. In this mode the pivot point can be set to an object that is mounted on the stage for example. The SmarPod always rotates around this object independent from translations in *X*, *Y* or *Z*. If the object is globally fixed, i.e. not connected with the stage, the *fixed* mode should be used.

Remember, that the pivot point (like speed, acceleration and most other settings) remains set only as long as the SmarPod is initialized. When the SmarPod is released and initialized again or when another program takes control of the SmarPod, the pivot point is reset to (0,0,0). To continue working with the original pivot point, the program(s) must take steps to memorize and restore it.

#### Computation of the Pose from Parameters

The parameters that are passed to the `Smarpod_Move` command or returned by `Smarpod_GetPose` are a struct of type `Smarpod_Pose` which is a 6-tuple ($X,Y,Z,\Theta_x,\Theta_y,\Theta_z$) that describes an absolute pose of the SmarPod. Rotations follow the “*X-Y-Z convention*” for Euler angles: first rotate around the *X* axis to the absolute angle $\Theta_x$, then around the *Y* axis to the absolute angle $\Theta_y$, then around the *Z* axis to absolute angle $\Theta_z$.

The pivot mode influences the order in which translation and rotation are applied (only mathematically – physically, rotation and translation are performed as one movement):

- Pivot mode *relative*: rotation comes before translation.
- Pivot mode *fixed*: translation comes before rotation.

**Readjusting Pose-Parameters and Physical Pose**

When the pivot point changes, the physical pose and the pose that was set with the last Move command do no longer match because the interpretation of the pose value depends on the pivot point, as can be seen in Figure 7. For example, let's assume that the pivot is (0,0,0) and the SmarPod has been moved to *X*=2mm and $\Theta_z$=5°. If the pivot is now changed to (500mm,0,0) the pose that `Smarpod_GetPose` returns would be about *X*=97µm, *Y*=43mm and $\Theta_z$=5°. There are two ways to make pose data and physical pose consistent:

- Update the pose data in your software with `Smarpod_GetPose` so it does represent the SmarPods current physical pose again.
- Move the SmarPod again with the same pose parameters. It is possible that the pose is no longer reachable because of the changed pivot.

### 2.5.2 Velocity

The movement velocity is set with `Smarpod_SetSpeed`. This function also enables or disables the speed-control mode. Speed-control is enabled after `Smarpod_Initialize` with a speed of 1mm/s.

- If *speed-control* is enabled, the stage moves smoothly from the start pose to the end pose. Depending on the start pose, end pose and the pivot point, the positioners move with different velocities. The *speed* parameter of `Smarpod_SetSpeed` determines the velocity of the fastest moving positioner in meters per second. The other positioners move slower or with the same velocity.
- If *speed-control* is disabled the *speed* parameter is ignored. All positioners are driven with the current maximum frequency (see `Smarpod_SetMaxFrequency`), so they move with approximately the same velocity. In this mode the stage does not necessarily move smoothly from the start to the end pose because some positioners may reach their target positions before others. This can tilt the stage temporarily during the movement.

  Movements with disabled speed-control less predictable than those with speed-control enabled.

The speed modes can be easily distinguished by the sound that is produced by the moving positioners. With speed-control enabled, the positioners change the driving frequency very fast which results in a noisy sound. Movements without speed-control use fixed frequencies which is also true for the sound. If the frequency is very high, movements with no speed-control may be difficult to hear. Some movements during calibration and reference-finding are performed without speed-control and use the frequency that was set with `Smarpod_SetMaxFrequency`.

If the highest reachable velocity is lower than the specified velocity, the SmarPod will move slower, but does not always report this as an error. However, if the speed parameter is set much higher than the possible velocity, this can cause false detection of blockages, which may be reported as `SMARPOD_ENDSTOP_REACHED_ERROR`. In such a case the SmarPod does not reach the target pose. The speed should then be set to a lower value.

The *maximum frequency* (see `Smarpod_SetMaxFrequency`) influences the reachable velocity. If the controller needs to drive a positioner with a frequency that would exceed the max. frequency, the actual frequency and the speed of this positioner are capped.

### 2.5.3 Waiting for Move-Completion

It is often necessary that a program knows when the SmarPod has reached the target position of a movement. This can be accomplished by calling `Smarpod_Move` with parameter `waitForCompletion` set to 1. In this mode the function returns when the movement has reached the target pose or an error has occurred. A typical application for this mode is the automatic scanning of an n-dimensional grid where a program moves the SmarPod to a point on the grid and measures a sensor signal. It is important that the measurement does not start before the SmarPod has reached its target pose. Note, that no other command can be executed (in the same thread) while the move command is waiting for completion. However, the program can call other SmarPod functions during the execution of a waiting move command from other threads.

`Smarpod_Move` can be called with `waitForCompletion` = 0 to return control to the caller immediately. In this mode, to wait for movement completions, the program must actively check the SmarPods move status (`Smarpod_GetMoveStatus`) and/or its current pose (`Smarpod_GetPose`). Since `Smarpod_Move` is not blocking the thread, the functions can be called from the same thread.

If a movement is started with `waitForCompletion` = 0 the caller will not be notified when the movement fails. To check that the SmarPod has reached the target pose when it has stopped, the current pose can be compared to the pose argument used in the call to `Smarpod_Move`, for example:

```
Smarpod_Move(id,&target,holdtime,0);
...
/* when the SmarPod has stopped */
Smarpod_GetPose(id,&pose);
bool ok = abs(pose.positionX - target.positionX) < tolerancePosition
       && abs(pose.positionY - target.positionY) < tolerancePosition
       && abs(pose.positionZ - target.positionZ) < tolerancePosition
       && abs(pose.rotationX - target.rotationX) < toleranceRotation
       && abs(pose.rotationY - target.rotationY) < toleranceRotation
       && abs(pose.rotationZ - target.rotationZ) < toleranceRotation;
```

### 2.5.4 Acceleration

The acceleration of movements can be limited optionally with `Smarpod_SetAcceleration`. If acceleration-control is enabled the SmarPod increases the movement speed with the specified acceleration. Before it reaches the destination pose, it decelerates with the same rate. Depending on the acceleration and speed values this can take between milliseconds and several seconds. Acceleration-control is disabled by default.

Like the movement velocity setting (see above), the acceleration parameter defines the acceleration of the fastest moving positioner. The other positioners have a lower or equal acceleration. As a consequence, the parameter does not specify the acceleration of a certain point of the stage but the maximum acceleration. Every point on the stage has an acceleration between zero and the acceleration parameter. Points outside the stage radius can have higher accelerations.

> Accelerated movements must only be started when the SmarPod does not move. It is not recommended to start an accelerated movement and overwrite it with another move command.

The acceleration-control can be disabled when calling `Smarpod_SetAcceleration`. If disabled, the SmarPod positioners start and stop as fast as possible. Acceleration-control depends on speed-control (see above). If speed-control is disabled the acceleration settings have no effect.

Acceleration is only supported by newer MCS firmware. If the SmarPod's MCS does not support it, the acceleration functions return with `SMARPOD_FEATURE_UNAVAILABLE_ERROR`.

**Stopping movements with enabled acceleration-control:**

When the `Smarpod_Stop` command is called once while a SmarPod moves with enabled acceleration-control, it begins to decelerate. It can take several seconds until the stage finally stops if the acceleration is small. This may or may not be intended by the user. To stop the SmarPod immediately (risking a higher acceleration) the stop command can be called a second time during the deceleration phase. See Figure 8: Stopping Accelerated Movements.

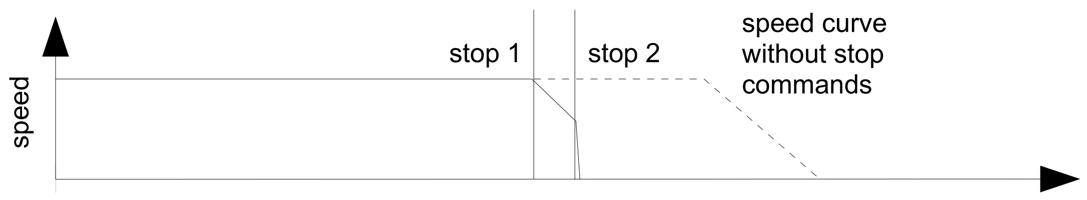


*Figure 8: Stopping Accelerated Movements*

### 2.5.5 Deviations from Perfect Trajectories

The stage's trajectory can deviate from a perfect trajectory if the movement includes a rotation. This is due to fact that the positioners of the SmarPod move with constant velocities (except if acceleration is enabled).

For example, when a SmarPod is moved from -10° to +10° around the *Z* axis (the other pose parameters are assumed to be 0 here), the stage moves slightly upwards and then downwards during the rotation. When the destination pose is reached, the *Z* translation of the stage is 0 again.

The deviation grows with the rotation angle. It can be reduced if the movement is sub-divided into smaller movements each moving the stage a small step towards the destination pose. In the example above the deviation can be significantly reduced if the movement from -10° to 10° degrees is performed in *N* steps with 20/*N* degrees.

# 3 Troubleshooting

All commands return with a status code that indicates either success or that the command could not be executed or has failed during execution. See 6.3 “Function Status Codes” for a list of status codes. Some codes indicate that the SmarPod state does not fulfill the requirements for a command. For example, movement commands return with `SMARPOD_NOT_REFERENCED_ERROR` if the SmarPod has not been referenced. Other codes indicate more severe errors. It is recommended to check the return status of all called SmarPod API commands to avoid problems.

## 3.1 System Configuration Error

`Smarpod_Open` and `Smarpod_Initialize` can return with the status code `SMARPOD_SYSTEM_CONFIGURATION_ERROR` which indicates that the configuration of the MCS controller does not match the SmarPod hardware model passed to the initialization function. This can simply mean that a wrong model code has been used which should be checked first. The error code can be returned after an update of the MCS controller firmware which has reset the internal configuration.

To configure the MCS use function `Smarpod_ConfigureSystem`. After calling it, a calibration of the positioners, followed by a referencing is recommended.

`SMARPOD_SYSTEM_CONFIGURATION_ERROR` is a special error insofar as **initialization has not failed** when this code is returned. Some functions, e.g. `Smarpod_ConfigureSystem`, can be called but most functions like `Smarpod_FindReferenceMarks`, `Smarpod_Move` or `Smarpod_GetPose` will also return with `SMARPOD_SYSTEM_CONFIGURATION_ERROR`. Do not forget to close/release a SmarPod whose initialization has returned with this error, to free the allocated resources!

## 3.2 Movement Failures Caused By Overloading

The power consumption increases with the number of positioners that move simultaneously and with the frequency they are driven with. Some SmarPods have stronger positioners which need more power than other models. If the consumed power exceeds the capacity of the MCS, move commands (including `Smarpod_FindReferenceMarks`) might return with `SMARPOD_OTHER_ERROR` or `SMARPOD_ENDSTOP_REACHED_ERROR` even if they are not blocked physically.

If you suspect an overload problem, please check if the MCS is connected to a power-supply that SmarAct recommends for your SmarPod model. If the right power-supply is connected and overload problems still occur you should lower the speed and/or the maximum frequency used for movements.

For `Smarpod_FindReferenceMarks` you can configure the behavior of the function so that fewer positioners are moved simultaneously (see section 2.4.5 “Finding Reference Marks“).

## 3.3 Communication Failures

If `SMARPOD_COMMUNICATION_ERROR` is returned by any command, the connection from the PC to the MCS controller is broken. When this error has been detected, most commands will return the same error code until the software is reset to an operational state.

To reset the SmarPod the software must re-initialize the MCS and the SmarPod. An application that gets the above error code from a command should stop calling SmarPod commands and possibly show an error message to the user who should check the physical connection and re-initialize the SmarPod.

## 3.4 Program Does Not Terminate

If the SmarPod library is not released before the program quits, it is possible that the program hangs for an unusally long time, possibly forever, before it terminates. You should call `Smarpod_Close` that has been initialized with `Smarpod_Open` and `Smarpod_ReleaseAll` if the program used `Smarpod_Initialize` for initialization.

# 4 Data Types

## Smarpod_Pose

```
typedef struct
{
    double positionX;
    double positionY;
    double positionZ;
    double rotationX;
    double rotationY;
    double rotationZ;
}Smarpod_Pose;
```

Describes the pose of a SmarPod.

- *positionX, ...Y and ...Z* – Translation of the stage in *X*,*Y* and *Z* direction (in meters).
- *rotationX, ...Y and ...Z* – Rotation angles of the stage around the *X*, *Y* and *Z*-axes (in degrees).

See also 2.5.1 - "Pose and Pivot Point".

See also `Smarpod_Move`, `Smarpod_IsPoseReachable`, `Smarpod_GetPose`.

# 5 Function Reference

## *Smarpod_GetDllVersion*

**Interface:**

```
Smarpod_Status Smarpod_GetDllVersion(unsigned int *major,
                                     unsigned int *minor,
                                     unsigned int *update);
```

**Description:**

This function returns the version number of the loaded SmarPod library. It can be called before initializing controllers and SmarPods.

**Parameters:**

- *major, minor, update* (unsigned 32 bit), output – Pointers to unsigned integer variables that the function writes the version number data to.

**Example:**

```
unsigned int major,minor,update;
Smarpod_GetDllVersion(&major,&minor,&update);
printf("using SmarPod library version %u.%u.%u\n",major,minor,update);
```

## *Smarpod_GetStatusInfo*

**Interface:**

```
Smarpod_Status Smarpod_GetStatusInfo(Smarpod_Status status,
                                     const char **statusInfo);
```

**Description:**

This function returns a textual description for a SmarPod status code. See section 6.3 “Function Status Codes” for a list of SmarPod status codes.

**Parameters:**

- *status* (Smarpod_Status), input – The status code.
- *statusInfo* (pointer to C string), output – The returned pointer to a const C string that contains the status description. The string is static, don't free or delete it.

**Example:**

```
Smarpod_Status result;
const char *info;
result = Smarpod_SetMaxFrequency(smarpodId,999999);  /* invalid frequency */
if(result != SMARPOD_OK)
{
   result = Smarpod_GetStatusInfo(result,&info);
   if(result == SMARPOD_OK)
      printf("Error while setting the SmarPod max. frequency: %s",info);
   else
      printf("Error: could not get status code info.");
}
```

## *Smarpod_GetModels*

**Interface:**

```
Smarpod_Status Smarpod_GetModels(unsigned int *modelList,
                                 unsigned int *ioListSize);
```

**Description:**

This function returns a list of all model codes supported by the SmarPod library.

**Parameters:**

- *modelList* (an array of unsigned int), output – An array the model codes will be stored in.
- *ioListSize* (pointer to unsigned int), input, output – The caller must pass the maximum size of the modelList array. When the function returns with no error *ioListSize* contains the actual number of codes written into the array. If the array passed is too small, the function returns with `SMARPOD_INVALID_PARAMETER_ERROR` and writes the required array size to **ioListSize.*

**Example:**

```
unsigned int models[128];
unsigned int listSize = 128;
Smarpod_Status result;
result = Smarpod_GetModels(models,&listSize);
if(result == SMARPOD_OK) {
   /* use the list... */
}
```

## *Smarpod_GetModelName*

**Interface:**

```
Smarpod_Status Smarpod_GetModelName(unsigned int model,
                                    const char **name);
```

**Description:**

This function returns the SmarPod model name for the code passed in *model*.

**Parameters:**

- *model* (unsigned int), input – A SmarPod model code.
- *name* (pointer to a constant string), output – The returned pointer points to a constant name string.

**Example:**

```
const char *name;
Smarpod_Status result;
result = Smarpod_GetModelName(10001,&name);
if(result == SMARPOD_OK) {
   printf("Name for SmarPod model code 10001 is: %s",name);
}
```

## Smarpod_Open

**Interface:**

```
Smarpod_Status Smarpod_Open(unsigned int *smarpodId,
                            unsigned long hardwareModel,
                            const char *locator,
                            const char *options);
```

**Description:**

This function initializes one SmarPod. The MCS that controls the SmarPod must be specified in the text field *locator*. On success the function returns `SMARPOD_OK` as the result and an internally generated ID for the initialized SmarPod in *smarpodId*.

In contrast to the (older and deprecated) method of initializing a SmarPod with `Smarpod_InitSystems` and `Smarpod_Initialize` the initialization of the MCS controller is handled internally by `Smarpod_Open`.

The caller must pass the hardware-model code of the SmarPod. A list of model codes for SmarAct hardware can be found in the document *SmarAct Hardware Codes.txt*.

| |
|---|
| Please ensure that the right model code is used. Otherwise the initialization function might succeed but the execution of commands can lead to undefined behavior. |

SmarPods that have been initialized with `Smarpod_Open` must be released with `Smarpod_Close`. `Smarpod_ReleaseAll` does not release SmarPods that have been initialized with `Smarpod_Open`.

Initialization can fail with an `SMARPOD_SYSTEM_CONFIGURATION_ERROR` error if the configuration of the MCS does not match the hardware model. In that case the intialization has not failed. The returned *smarpodId* is valid and can be used to call function `Smarpod_ConfigureSystem` to configure the MCS. Calling other functions would result in the same error. Remember that an initialization that has returned with this error must be closed! See section 3.1 “System Configuration Error” for more information.

See section 2.4 “Initialization” for more information.

**Parameters:**

- *smarpodId* (unsigned 32 bit), output – an ID for the SmarPod. The ID must be passed to all functions that address the SmarPod.
- *hardwareModel* (unsigned 32 bit), input – the hardware model code (see text).
- *locator* (C string), input – the system locator that identifies the MCS controller.
- *options* (C string), input – options for the SmarPod initialization (currently unused – pass an empty string for now).

**Example:**

```
unsigned long smarpodId;          /* a variable to store SmarPod ID generated by Open */
unsigned long hwModel = 10001;    /* specifies the SmarPod 110.45 S (nano) */
Smarpod_Status result;

result = Smarpod_Open(&smarpodId,hwModel,”usb:id:123456789”,””);

if (result == SMARPOD_SYSTEM_CONFIGURATION_ERROR) {
   // the MCS configuration of the MCS does not match the selected SmarPod model.
   // ...handle this error...
   // don't forget to close the SmarPod...
   result = Smarpod_Close(smarpodId);
} else if (result == SMARPOD_OK) {
   // ...work with the SmarPod...

   result = Smarpod_Close(smarpodId);
}
```

## ***Smarpod_Close***

**Interface:**

```
Smarpod_Status Smarpod_Close(unsigned int smarpodId);
```

**Description:**

This function releases the SmarPod *smarpodId.*

Only SmarPods that have been initialized with `Smarpod_Open` can be closed with this function.

> It is important to call `Smarpod_Close` for each SmarPod that has been initialized with `Smarpod_Open`. Not closing a SmarPod will cause a resource leak. An attempt to open an unclosed SmarPod again can fail because some resources (e.g. the USB or network connection to the MCS ) are still hold by the previous initialization.

**Parameters:**

- *smarpodId* (unsigned int), input – The SmarPod ID

**Example:**

See example for `Smarpod_Open`.

## *Smarpod_FindSystems*

**Interface:**

```
Smarpod_Status Smarpod_FindSystems(const char *options,
                                   char *outBuffer,
                                   unsigned int *ioBufferSize);
```

**Description:**

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

**Parameters:**

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

**Example:**

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

## *Smarpod_GetSystemLocator*

**Interface:**

```
SA_STATUS Smarpod_GetSystemLocator(SA_INDEX smarpodId,
                                   char *outBuffer,
                                   unsigned int *ioBufferSize);
```

**Description:**

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

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – ID of an initialized SmarPod.
- *outBuffer* (char pointer), output – Pointer to a buffer which holds the locator after the function has returned
- *ioBufferSize* (unsigned int pointer), input/output – Specifies the size of *outBuffer* before the function call. After the function call it holds the number of bytes written to *outBuffer.*

**Example:**

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

## *Smarpod_InitSystems*

This function has been deprecated. Please use the functions `Smarpod_Open` and `Smarpod_Close` for initialization and release. This function may be removed from the API in the future!

**Interface:**

```
Smarpod_Status Smarpod_InitSystems(const unsigned int *initList,
                                   unsigned int initListSize);
```

**Description:**

This function initializes MCS controller systems. It must be called before SmarPods can be initialized with `Smarpod_Initialize`.

When using `Smarpod_Open` for initialization (as is recommended beginning with API version 1.5) `Smarpod_InitSystems` does not have to be called.

**Parameters:**

- *initList* (array of unsigned 32 bit), input – an array of MCS System IDs. If the list has a size of 1 and the value is 0, the function initializes the first MCS it can find.
- *InitListSize* (unsigned 32 bit), input – the number of elements in *initList*.

**Example:**

```
Smarpod_Status result;
const unsigned int initList[2] = {3981737919, 3341233441};
result = Smarpod_InitSystems(initList,2);
if (result == SMARPOD_OK) {
   // systems have been successfully acquired
}
```

## Smarpod_Initialize

This function has been deprecated. Please use the functions `Smarpod_Open` and `Smarpod_Close` for initialization and release. This function may be removed from the API in the future!

**Interface:**

```
Smarpod_Status Smarpod_Initialize(unsigned int smarpodId,
                                   unsigned long hardwareModel,
                                   unsigned int systemId);
```

**Description:**

This function initializes a SmarPod. The caller chooses a *SmarPod ID* which is used to address the SmarPod when calling SmarPod functions later. The ID must be an unsigned integer number in the range 0 to *SMARPOD_MAX_ID* (included).

Before `Smarpod_Initialize` can be called, the MCS controller(s) must have been initialized.

The caller must pass the hardware-model code of the connected SmarPod. A list of model codes for SmarAct hardware can be found in the document *SmarAct Hardware Codes.txt*.

Ensure that the right model code is used. Otherwise the initialization function might succeed but the execution of commands can lead to undefined behavior.

Initialization can fail with an `SMARPOD_SYSTEM_CONFIGURATION_ERROR` error. See section 3.1 “System Configuration Error” for more information.

See section 2.4 “Initialization” for more information.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – an ID for the SmarPod. Can be choosen by the caller. The ID must be passed to all functions that address the SmarPod.
- *hardwareModel* (unsigned 32 bit), input – the hardware model code (see text).
- *systemId* (unsigned 32 bit), input – the ID of the MCS controller the SmarPod is connected to.

**Example:**

```
unsigned long smarpodId = 0;
unsigned long hwModel = 10001;    /* specifies the SmarPod 110.45 S (nano) */
unsigned int mcsId = 1234512345;
Smarpod_Status result;
result = Smarpod_Initialize(smarpodId,hwModel,mcsId);
if (result == SMARPOD_OK) {
   // SmarPod has been successfully initialized
}
```

## *Smarpod_ReleaseAll*

This function has been deprecated. Please use the functions `Smarpod_Open` and `Smarpod_Close` for initialization and release. This function may be removed from the API in the future!

**Interface:**

```
Smarpod_Status Smarpod_ReleaseAll();
```

**Description:**

This function releasesall SmarPods that have been initialized with `Smarpod_Initialize` and all MCS controllers that have been initialized with `Smarpod_InitSystems` or with `SA_InitSystems` from the MCS API. `Smarpod_ReleaseAll` does not stop the released SmarPod. If the positioners are in holding-mode when quitting a program and calling `Smarpod_ReleaseAll`, they stay in that mode and the SmarPod will keep it's last pose until the MCS is switched off or another program continues controlling the SmarPod.

`Smarpod_ReleaseAll` does **not** release SmarPods that have been initialized with `Smarpod_Open` and it does not release MCS that have been initialized with `SA_OpenSystem` from the MCS API.

**Example:**

```
Smarpod_Status result;
const unsigned int initList[2] = {3981737919, 3341233441};
result = Smarpod_InitSystems(initList,2);
if (result == SMARPOD_OK) {
   result = Smarpod_ReleaseAll();
}
```

## *Smarpod_ConfigureSystem*

**Interface:**

```
Smarpod_Status Smarpod_ConfigureSystem(unsigned int smarpodId);
```

**Description:**

This function configures the MCS for a certain SmarPod model.

> It is recommended to call this function once after installing a new firmware on the MCS controlling the SmarPod. Installing firmware resets the configuration to default values which may not be correct for the SmarPod model. After calling this function the sensor-calibration of the SmarPod positioners is not longer valid. So a call to this function should be followed by a calibration.

See also 2.4.1 “Initializing a SmarPod” and 3.1 “System Configuration Error”.

See also `Smarpod_Open`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.

**Example:**

```
unsigned int smarpodId;
Smarpod_Status result;
result = Smarpod_Open(&smarpodId,hwModel,”usb:id:123456789”,””);
if (result == SMARPOD_SYSTEM_CONFIGURATION_ERROR) {
   result = Smarpod_ConfigureSystem(smarpodId);
   if(result == SMARPOD_OK)
   {
      result = Smarpod_Calibrate(smarpodId);
      if(result != SMARPOD_OK)
         exit(1);
   }
} else if(result != SMARPOD_OK) {
   exit(1);
}

int referenced;
result = Smarpod_IsReferenced(smarpodId,&referenced);
if(result != SMARPOD_OK) {
   exit(1);
}
if(referenced == 0) {
   result = Smarpod_FindReferenceMarks(smarpodId);
   if(result != SMARPOD_OK) {
      exit(1);
   }
}

result = Smarpod_Close(smarpodId);
```

## *Smarpod_Set_...*

### *Smarpod_Set_ui, Smarpod_Set_i, Smarpod_Set_d*

**Interface:**

```
Smarpod_Status Smarpod_Set_ui(unsigned int smarpodId,
                              unsigned int property,
                              unsigned int value);
Smarpod_Status Smarpod_Set_i(unsigned int smarpodId,
                              unsigned int property,
                              int value);
Smarpod_Status Smarpod_Set_d(unsigned int smarpodId,
                              unsigned int property,
                              double value);
```

**Description:**

These functions write a property of the SmarPod. The *property* argument selects the property which is set to *value*. Different functions are available for different value data types.

See also `Smarpod_Get_...`.

See section 2.3 “Properties” for an introduction to properties.

See appendix 6.2 “SmarPod Properties And Values” for a list of properties and their values.

**Example:**

```
Smarpod_Status s;
s = Smarpod_Set_ui(0,SMARPOD_FREF_METHOD,SMARPOD_METHOD_ZSAFE);
```

## *Smarpod_Get_...*

### *Smarpod_Get_ui, Smarpod_Get_i, Smarpod_Get_d*

**Interface:**

```
Smarpod_Status Smarpod_Get_ui(unsigned int smarpodId,
                              unsigned int property,
                              unsigned int *value);
Smarpod_Status Smarpod_Get_i(unsigned int smarpodId,
                              unsigned int property,
                              int *value);
Smarpod_Status Smarpod_Get_d(unsigned int smarpodId,
                              unsigned int property,
                              double *value);
```

**Description:**

These function read a property of the SmarPod. The *property* argument selects the property. Its value is returned in *value*. Different functions are available for different value data types.

See also `Smarpod_Set_...`.

See section 2.3 “Properties” for an introduction to properties.

See appendix 6.2 “SmarPod Properties And Values” for a list of properties and their values.

**Example:**

```
Smarpod_Status result;
unsigned int method;
result = Smarpod_Get_ui(0,SMARPOD_FREF_METHOD,&method);
```

## *Smarpod_Calibrate*

**Interface:**

```
Smarpod_Status Smarpod_Calibrate(unsigned int smarpodId);
```

**Description:**

This function calibrates all positioners of the SmarPod.

The function blocks the calling thread until all positioners have been calibrated or an error has occurred. It can be interrupted by `Smarpod_Stop` which must be called from a different thread. In this case the function returns with `SMARPOD_STOPPED_ERROR`.

See also section 2.4.4 “Calibrating the Sensors“.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.

**Example:**

```
Smarpod_Status result;
result = Smarpod_Calibrate(smarpodId);
```

## *Smarpod_FindReferenceMarks*

**Interface:**

```
Smarpod_Status Smarpod_FindReferenceMarks(unsigned int smarpodId);
```

**Description:**

This function performs some movements on the SmarPod to find the reference mark of each positioner. After the SmarPod initialization this function must be called once if the reference marks are not already known, which can be queried with `Smarpod_IsReferenced`. The reference data is stored in the controller until it is switched off. Therefore it is not necessary to find the reference marks at every program start.

Some properties can be set that configure the behavior of the function. See appendix 6.2 “SmarPod Properties And Values” for a list of properties.

The function blocks the calling thread until all reference marks are found or an error has occurred. It can be interrupted by `Smarpod_Stop` which must be called from a different thread. In this case the function returns with `SMARPOD_STOPPED_ERROR`.

See also `Smarpod_IsReferenced` and section 2.4.5 “Finding Reference Marks”.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.

**Example:**

```
Smarpod_Status s;
s = Smarpod_Set_ui(0,SMARPOD_FREF_METHOD,SMARPOD_METHOD_ZSAFE);
s = Smarpod_Set_ui(0,SMARPOD_FREF_ZDIRECTION,SMARPOD_NEGATIVE);
s = Smarpod_Set_ui(0,SMARPOD_FREF_AND_CAL_FREQUENCY,2000);
s = Smarpod_FindReferenceMarks(0);
```

## *Smarpod_IsReferenced*

**Interface:**

`Smarpod_Status Smarpod_IsReferenced(unsigned int smarpodId, int *referenced);`

**Description:**

This function returns the referenced-state of the SmarPod. If *referenced* is 1, the controller knows the positions of the positioners. If *referenced* is 0, it does not and `Smarpod_FindReferenceMarks` must be called before move commands can be used.

See also `Smarpod_FindReferenceMarks`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *referenced* (signed 32 bit), output – The returned referenced-state. 0: not referenced, 1: referenced.

**Example:**

```
Smarpod_Status s;
int referenced;
s = Smarpod_IsReferenced(smarpodId,&referenced);
if(s == SMARPOD_OK) {
   if(referenced == 0)
   {
      s = Smarpod_FindReferenceMarks(smarpodId);
   }
}
```

## *Smarpod_SetSensorMode*

**Interface:**

```
Smarpod_Status Smarpod_SetSensorMode(unsigned int smarpodId,
                                     unsigned int mode);
```

**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. If other SmarPods or positioners are connected to the same controller, this command sets the mode for their sensors too. Please refer to section 2.2 “Sensor Modes“ for more information.

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

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

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *mode* (unsigned 32 bit), input – The new sensor mode. Must be one of `SMARPOD_SENSORS_DISABLED`, `SMARPOD_SENSORS_ENABLED`, `SMARPOD_SENSORS_POWERSAVE`.

**Example:**

```
Smarpod_Status result;
result = Smarpod_SetSensorMode(smarpodId,SMARPOD_SENSORS_ENABLED);
```

## ***Smarpod_GetSensorMode***

**Interface:**

```
Smarpod_Status Smarpod_GetSensorMode(unsigned int smarpodId,
                                     unsigned int *mode);
```

**Description:**

This function returns the current sensor mode of the positioners connected to the SmarPod's controller.

See also `Smarpod_SetSensorMode` and section 2.2 “Sensor Modes“.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *mode* (unsigned 32 bit), output – The returned sensor mode.

**Example:**

```
Smarpod_Status result;
unsigned int mode;
result = Smarpod_GetSensorMode(smarpodId,&mode);
```

## *Smarpod_SetMaxFrequency*

**Interface:**

```
Smarpod_Status Smarpod_SetMaxFrequency(unsigned int smarpodId,
                                        unsigned int frequency);
```

**Description:**

This function may be used to define the maximum frequency that the positioners are driven with. The frequency influences the maximum velocity that can be reached by the SmarPod. If the controller needs to drive a positioner with a certain frequency and that frequency exceeds the max. frequency, the actual frequency and the velocity are capped.

See section 2.5 “Moving the SmarPod” for more information about the relationship between maximum frequency and movements.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *frequency* (unsigned 32 bit), input – The new maximum frequency for all positioners of the SmarPod (1 to 18500).

**Example:**

```
Smarpod_Status result;
result = Smarpod_SetMaxFrequency(smarpodId,3000);
```

## *Smarpod_GetMaxFrequency*

**Interface:**

```
Smarpod_Status Smarpod_GetMaxFrequency(unsigned int smarpodId,
                                       unsigned int *frequency);
```

**Description:**

This function returns the current maximum frequency the SmarPod's positioners are allowed to use.

See also `Smarpod_SetMaxFrequency`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *frequency* (unsigned 32 bit), output – The returned frequency.

**Example:**

```
Smarpod_Status result;
unsigned int frequency;
result = Smarpod_GetMaxFrequency(smarpodId,&frequency);
```

## *Smarpod_SetSpeed*

**Interface:**

```
Smarpod_Status Smarpod_SetSpeed(unsigned int smarpodId,
                                int speedControl,
                                double speed);
```

**Description:**

This function sets the movement velocity. It can set two different movement modes:

| speedControl | Movement |
|---|---|
| *0 (off)* | Speed-control is disabled. The *speed* parameter has no effect here. The actual speed depends on the maximum frequency which can be set with `Smarpod_SetMaxFrequency`. All positioners move with the same driving frequency. |
| *1 (on)* | Speed-control is enabled. The positioners move with individual speeds to perform a smooth movement from the start to the end pose. The *speed* parameter determines the speed of the fastest moving positioner. |

See section 2.5 “Moving the SmarPod” for more information about SmarPod movements and the effect of the different speed modes.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *speedControl* (signed 32 bit), input – Enables (1) or disables (0) speed-control.
- *speed* (double precision floating point), input – The velocity in meters per second. The maximum speed depends on the capabilities of the device and its components. Maximum reachable speed is typically 1 to 5 mm/seconds but this can vary (see section 2.5 “Moving the SmarPod”).

**Example:**

```
Smarpod_Status result;
result = Smarpod_SetSpeed(smarpodId,1,0.001);  /* sets speed to 1 mm/sec */
```

## *Smarpod_GetSpeed*

**Interface:**

```
Smarpod_Status Smarpod_GetSpeed(unsigned int smarpodId,
                                int *speedControl,
                                double *speed);
```

**Description:**

This function returns the current speed-control and speed settings.

See also `Smarpod_SetSpeed`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *speedControl* (signed 32 bit), output  – The returned speed-control mode. 0: disabled, 1: enabled.
- *speed* (double precision floating point), output – The returned speed that was set with `Smarpod_SetSpeed` in meters per second.

**Example:**

```
Smarpod_Status result;
int speedControl;
double speed;
result = Smarpod_GetSpeed(smarpodId,&speedControl,&speed);
```

## Smarpod_SetAcceleration

**Interface:**

```
Smarpod_Status Smarpod_SetAcceleration(unsigned int smarpodId,
                                int accelControl,
                                double acceleration);
```

**Description:**

This function sets the movement acceleration. If acceleration-control is not supported by the MCS controller that drives the SmarPod the function returns with an error.

See section 2.5.4 “Acceleration” for more information.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *accelControl* (signed 32 bit), input – Enables (1) or disables (0) acceleration-control.
- *acceleration* (double precision floating point), input – The acceleration in m/s$^2$ (1x10$^{-6}$ to 10).

**Example:**

```
Smarpod_Status result;
result = Smarpod_SetAcceleration(smarpodId,1,0.001);  /* sets accel. to 1 mm/sec^-2 */
```

## *Smarpod_GetAcceleration*

**Interface:**

```
Smarpod_Status Smarpod_GetAcceleration(unsigned int smarpodId,
                              int *accelControl,
                              double *acceleration);
```

**Description:**

This function returns the current acceleration-control and acceleration settings. When the SmarPod is initialized, this function can be used to check whether acceleration-control is supported by the MCS that drives the SmarPod. If it is not supported, the function returns with an error.

See section 2.5.4 “Acceleration” for more information.

See also `Smarpod_SetAcceleration`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *accelControl* (signed 32 bit), output – The acceleration-control mode. 0: disabled, 1: enabled.
- *acceleration* (double precision floating point), output – The acceleration in $m/s^2$ that was set with `Smarpod_SetAcceleration`.

**Example:**

```
Smarpod_Status result;
int accelControl;
double accel;
result = Smarpod_GetAcceleration(smarpodId,&accelControl,&accel);
if(result == SMARPOD_FEATURE_NOT_AVAILABLE_ERROR)
   printf(“acceleration-control not available”);
else if(accelControl == 0)
   printf(“acceleration-control is disabled”);
else
   printf(“the current movement acceleration: %lf”,accel);
```

## *Smarpod_SetPivot*

**Interface:**

```
Smarpod_Status Smarpod_SetPivot(unsigned int smarpodId,
                                const double *pivot);
```

**Description:**

This function sets the virtual pivot point for a SmarPod. The pivot point ($P_x$,$P_y$,$P_z$) is the center of SmarPod rotations.

See also `Smarpod_GetPivot`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *pivot* (pointer to an array of 3 doubles), input – Contains the three cartesian coordinates of pivot point [$P_x$,$P_y$,$P_z$]. All coordinates $P_{x,y,z}$ are in meters.

**Example:**

```
Smarpod_Status result;
double pivot[3];
pivot[0] = 0.0; pivot[1] = 0.0; pivot[2] = 0.015;
result = Smarpod_SetPivot(smarpodId,pivot);
```

## ***Smarpod_GetPivot***

**Interface:**

```
Smarpod_Status Smarpod_GetPivot(unsigned int smarpodId,
                                double *pivot);
```

**Description:**

This function returns the current virtual pivot point.

See also `Smarpod_SetPivot`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *pivot* (pointer to an array of 3 doubles), output – The returned current pivot point [$P_x$,$P_y$,$P_z$]. Must be a pointer to an array that can store 3 doubles. All coordinates $P_{x,y,z}$ are in meters.

**Example:**

```
Smarpod_Status result;
double pivot[3];
result = Smarpod_GetPivot(smarpodId,pivot);
```

## *Smarpod_IsPoseReachable*

**Interface:**

```
Smarpod_Status Smarpod_IsPoseReachable(unsigned int smarpodId,
                                        const Smarpod_Pose *pose,
                                        int *reachable);
```

**Description:**

With is function it can be tested if the SmarPod can reach a pose. The function does not move the SmarPod. In contrast to `Smarpod_Move` the function does not return with an error if the pose is not reachable.

See also `Smarpod_Move`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- pose (pointer to struct Smarpod_Pose), input – The pose to test (in meters, degrees).
- *reachable* (unsigned 32 bit), output – The result of the test on reachability. 0: the pose cannot be reached, 1: the pose can be reached by the SmarPod.

**Example:**

```
Smarpod_Status result;
Smarpod_Pose pose;
int reachable;
result = Smarpod_IsPoseReachable(smarpodId,&pose,&reachable);
```

## *Smarpod_GetPose*

**Interface:**

```
Smarpod_Status Smarpod_GetPose(unsigned int smarpodId,
                               Smarpod_Pose *pose);
```

**Description:**

This function returns the current physical pose of the SmarPod. The SmarPod must have been referenced before calling this function.

The pose depends on the pivot point that is set when GetPose is called.

The command can be called while the SmarPod is moving.

See also 2.5.1 “Pose and Pivot Point”.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *pose* (pointer to struct Smarpod_Pose), output – The current physical SmarPod pose (in meters, degrees). If the function returns with an error, pose is undefined.

**Example:**

```
Smarpod_Status result;
Smarpod_Pose pose;
result = Smarpod_GetPose(smarpodId,&pose);
```

## *Smarpod_GetMoveStatus*

**Interface:**

```
Smarpod_Status Smarpod_GetMoveStatus(unsigned int smarpodId,
                                     unsigned int *moveStatus);
```

**Description:**

This function returns the current move-status of the SmarPod. It can be used to check if the SmarPod is moving, stopped or in holding-mode.

See also `Smarpod_Move`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- moveStatus (unsigned 32 bit), output – The current move-status. Possible values are `SMARPOD_STOPPED`, `SMARPOD_HOLDING`, `SMARPOD_MOVING`, `SMARPOD_CALIBRATING` and `SMARPOD_REFERENCING`.

**Example:**

```
Smarpod_Status result;
unsigned int moveStatus;
result = Smarpod_GetMoveStatus(smarpodId,&moveStatus);
if(result == SMARPOD_OK)
{
   switch(moveStatus) {
   case SMARPOD_STOPPED: printf("SmarPod is stopped"); break;
   case SMARPOD_HOLDING: printf("SmarPod is holding the pose"); break;
   case SMARPOD_MOVING: printf("SmarPod is moving"); break;
   case SMARPOD_CALIBRATING: printf("SmarPod is calibrating"); break;
   case SMARPOD_REFERENCING: printf("SmarPod is referencing"); break;
   default:
      printf("unknown SmarPod move-status");
   }
}
```

## *Smarpod_Move*

**Interface:**

```
Smarpod_Status Smarpod_Move(unsigned int smarpodId,
                             const Smarpod_Pose *pose,
                             unsigned int holdTime,
                             int waitForCompletion);
```

**Description:**

Moves the SmarPod to a new pose. If the destination pose is outside the reachble area of the SmarPod, the function returns `SMARPOD_POSE_UNREACHABLE_ERROR` and does not move the SmarPod.

The *holdTime* parameter specifies the time the SmarPod will actively hold the destination pose. Deviations from the pose are compensated by moving back to the pose. Holding is influenced by the sensor mode: in power save mode the sensors are not permanently enabled. They are pulsed with a few Hertz. The pose corrections are performed with that frequency.

If *waitForCompletion* is set and the SmarPod is blocked while moving, the function will return with `SMARPOD_ENDSTOP_REACHED_ERROR`. If `Smarpod_Stop` is called while `Smarpod_Move` is waiting for the movement to finish, `SMARPOD_STOPPED_ERROR` is returned.

`Smarpod_Move` will return with error `SMARPOD_MOVING_ERROR` if another move command was called with *waitForCompletion* = 1 (in a different thread) which has not completed yet. Therefore it is not possible to overwrite movements that wait for completion with another call to `Smarpod_Move`. However, it is possible and safe to call `Smarpod_Move` if a previous movement that has been started with *waitForCompletion* = 0 is still executing.

See also section 2 “Getting Started”.

See also `Smarpod_GetMoveStatus`.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *pose* (pointer to struct Smarpod_Pose), input – The pose to move to (in meters, degrees).
- *holdTime* (unsigned 32 bit), input – Specifies how long (in milliseconds) the positioners remain in *holding-mode* after reaching their target positions. The range is 0...60000. Pass `SMARPOD_HOLDTIME_INFINITE` to hold the pose indefinitely.
- *waitForCompletion* (signed 32 bit), input – If 1, the function does not return until all positioners have stopped moving. If 0, the function returns immediately.

**Example:**

```
Smarpod_Status result;
Smarpod_Pose pose;
result = Smarpod_Move(smarpodId,&pose,SMARPOD_HOLDTIME_INFINITE,1);
```

## *Smarpod_Stop*

**Interface:**

```
Smarpod_Status Smarpod_Stop(unsigned int smarpodId);
```

**Description:**

Stops SmarPod movements. If `Smarpod_Stop` is called when `Smarpod_FindReferenceMarks` or `Smarpod_Calibrate` are running, the functions return with `SMARPOD_STOPPED_ERROR`. `Smarpod_Move` does not return with `SMARPOD_STOPPED_ERROR` if stopped.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.

**Example:**

```
Smarpod_Status result;
Smarpod_Pose pose;
result = Smarpod_Move(smarpodId,&pose,SMARPOD_HOLDTIME_INFINITE,0);
if(result == SMARPOD_OK)
{
   Sleep(200);
   result = Smarpod_Stop(smarpodId);
}
```

## *Smarpod_StopAndHold*

**Interface:**

```
Smarpod_Status Smarpod_StopAndHold(unsigned int smarpodId,
                                   unsigned int holdTime);
```

**Description:**

Stops SmarPod movements. In contrast to `Smarpod_Stop`, which completely stops the position control, the last positions of the positioners are actively hold when `Smarpod_StopAndHold` is called and the SmarPod move status becomes `SMARPOD_HOLDING`.

If the function called when `Smarpod_FindReferenceMarks` or `Smarpod_Calibrate` are running, the functions return with `SMARPOD_STOPPED_ERROR`. `Smarpod_Move` does not return with `SMARPOD_STOPPED_ERROR` if stopped.

**Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *holdTime* (unsigned 32 bit), input – Specifies how long (in milliseconds) the positioners remain in *holding-mode* after reaching their target positions. The range is 0...60000. Pass `SMARPOD_HOLDTIME_INFINITE` to hold the pose indefinitely.

**Example:**

```
Smarpod_Status result;
Smarpod_Pose pose;
result = Smarpod_Move(smarpodId,&pose,SMARPOD_HOLDTIME_INFINITE,0);
if(result == SMARPOD_OK)
{
   Sleep(200);
   result = Smarpod_StopAndHold(smarpodId,SMARPOD_HOLDTIME_INFINITE);
}
```

# 6 Appendix

## 6.1 Code Example

```
#include <stdlib.h>
#include <stdio.h>
#include <SmarPod.h>

const int calibrate = 0;                        /* change to 1 to calibrate */
const unsigned int model = 10001;               /* SmarPod 110.45 S */
const char mcsLocator[] = “usb:id:123456789”; /* The locator of the SmarPods MCS */

const double xyMax = 0.0102;
const Smarpod_Pose pZero = { 0.,0.,0., 0.,0.,0. };
const Smarpod_Pose pZ0Left = { -xyMax,0.,0., 0.,0.,0. };
const Smarpod_Pose pZ0Right = { xyMax,0.,0., 0.,0.,0. };

const int enableAccelerationControl = 0;  /* change to 1 for acceleration control */

int main() {
   unsigned int major,minor,update;
   unsigned int initList[1];
   int referenced;
   unsigned int id;
   unsigned int status;
   Smarpod_Pose pose;

   Smarpod_GetDLLVersion(&major,&minor,&update);
   printf("using SmarPod library version %u.%u.%u\n",major,minor,update);

   if(Smarpod_Open(&id,model,mcsLocator,””) != SMARPOD_OK )
      return 1;

   Smarpod_SetSensorMode(id,SMARPOD_SENSORS_ENABLED);
   if(calibrate)
      if( Smarpod_Calibrate(id) != SMARPOD_OK )
         return 1;

   Smarpod_SetMaxFrequency(id,18500);   /* set 18.5kHz for speed-controlled movements */
   Smarpod_SetSpeed(id,1,0.002);        /* set to 2mm/s */

   Smarpod_Set_ui(id,SMARPOD_FREF_AND_CAL_FREQUENCY,8000);  /* set to 8kHz */
   Smarpod_IsReferenced(id,&referenced);
   if(!referenced)                      /* only if not referenced */
      if( Smarpod_FindReferenceMarks(id) != SMARPOD_OK )
         return 1;

   if(enableAccelerationControl)
      if( Smarpod_SetAcceleration(id,1,0.1) != SMARPOD_OK )
         printf("could not enable acceleration-control, error %u",status);

   Smarpod_Move(id,&pZ0Left,1000,1);
   Smarpod_SetSpeed(id,1,0.006);
   Smarpod_Move(id,&pZ0Right,1000,1);
   Smarpod_SetSpeed(id,1,0.01);
   Smarpod_Move(id,&pZero,SMARPOD_HOLDTIME_INFINITE,1); /* hold last pose forever */
   if( Smarpod_GetPose(id,&pose) != SMARPOD_OK )
      return 1;
   printf("pose = (%lf,%lf,%lf,%lf,%lf,%lf)\n",pose.positionX,pose.positionY,
      pose.positionZ,pose.rotationX,pose.rotationY,pose.rotationZ);

   Smarpod_Close(id);
   return 0;
}
```

## 6.2 SmarPod Properties And Values

This section lists the properties that can be set with the property set functions `Smarpod_Set_...`. The prefix "`SMARPOD_`" is omitted from the constants in this table, so instead of "`FREF_METHOD`" use "`SMARPOD_FREF_METHOD`". *Value Type* indicates which `Smarpod_Set_...` and `Smarpod_Get_...` function must be used to write and read the property. E.g., if the *Value Type* is *ui* use `Smarpod_Get_ui` and `Smarpod_Set_ui` to read and write the property.

| Property Name | Values | Description |
|---|---|---|
| `FREF_METHOD` (Value Type: ui) | Selects the reference mark search algorithm for function `Smarpod_FindReferenceMarks`. See section 2.4.5 "Finding Reference Marks". | |
| | `DEFAULT` | Selects the default method for the SmarPod model. |
| | `METHOD_SEQUENTIAL` | Selects the *Sequential* search algorithm. The positioners are referenced one at a time. |
| | `METHOD_ZSAFE` | Selects the *Z-Safe* search algorithm. Optimizes referencing for safe movements in Z direction. First the radial positioners are referenced together so that the stage moves up or down, depending on the direction property. Then the tangential positioners are referenced together. This method is only available for rotation-symmetric models except for SmarPods with micro-sensor positioners. |
| | `METHOD_XYSAFE` | Selects the *XY-Safe* search algorithm. The movement directions for the X and Y axes can be specified separately with the properties `FREF_XDIRECTION` and `FREF_YDIRECTION`. Only available for parallel SmarPods. |
| `FREF_XDIRECTION` `FREF_YDIRECTION` `FREF_ZDIRECTION` (Value Type: ui) | Properties to set the primary movement direction when referencing. | |
| | `DEFAULT` | Selects the default direction for the axis. |
| | `POSITIVE` | Specifies a positive movement direction. |
| | `NEGATIVE` | Specifies a negative movement direction. |
| | `REVERSE` | The *reverse-direction* flag can be *OR*'ed with the direction constant `POSITIVE` or `NEGATIVE` if the SmarPod has distance coded reference marks. |
| `FREF_AND_CAL_FREQUENCY` (Value Type: ui) | Frequency (in Hertz) used when referencing or calibrating the positioners. If this property is not set or set to 0, the frequency that was set with `Smarpod_SetMaxFrequency` is used for referencing and calibration. | |
| `PIVOT_MODE` (Value Type: ui) | Sets the behavior of the pivot point when the stage is moved linearly. | |
| | `PIVOT_RELATIVE` | The pivot point is relative to the stage position. When the stage moves by a certain offset, the pivot moves by the same offset. This mode is useful if the SmarPod should rotate around objects that are mounted on the stage. This is the default mode. |
| | `PIVOT_FIXED` | The pivot point is relative to the base plate of the SmarPod. When the stage moves, the pivot point does not. This mode is useful if the SmarPod should rotate around objects that are not mounted on the stage. |

## 6.3 Function Status Codes

`0: SMARPOD_OK`
The function call was successful.

`1: SMARPOD_OTHER_ERROR`
An uncategorized error has occurred.

`2: SMARPOD_SYSTEM_NOT_INITIALIZED_ERROR`
This error is returned if the initialization of the MCS controller fails in the call to `Smarpod_InitSystems` or if `Smarpod_Initialize` or `Smarpod_ReleaseAll` are called without an initialized MCS controller.

`3: SMARPOD_NO_SYSTEMS_FOUND_ERROR`
Returned by `Smarpod_InitSystems` if no MCS controller systems can be found.

`4: SMARPOD_INVALID_PARAMETER_ERROR`
Returned by various functions if a parameter value is invalid and if there is no error that is more specific.

`5: SMARPOD_COMMUNICATION_ERROR`
Can be returned by various functions if there is a communication problem with the MCS controller. See 3.3 “Communication Failures” for more information.

`6: SMARPOD_UNKNOWN_PROPERTY_ERROR`
Returned by property related function if an unknown property is passed.

`7: SMARPOD_RESOURCE_TOO_OLD_ERROR`
Returned if a recource the SmarPod (library) depends on is too old (e.g. the MCSControl library).

`8: SMARPOD_FEATURE_UNAVAILABLE_ERROR`
Returned if a certain feature is not supported by the soft- or hardware, e.g. acceleration-control.

`9: SMARPOD_INVALID_SYSTEM_LOCATOR_ERROR`
Returned by `Smarpod_Open` if the locator string has an wrong formatting.

`10: SMARPOD_QUERYBUFFER_SIZE_ERROR`
Returned by functions that write data in a binary or `char` buffer (e.g. `Smarpod_FindSystems`) if the user-supplied buffer is too small to hold the returned data.

`11: SMARPOD_COMMUNICATION_TIMEOUT_ERROR`
Returned if an expected answer from the MCS controller takes too long.

`12: SMARPOD_DRIVER_ERROR`
Returned if a driver that is needed for communicating with the controller could not be loaded.

`500: SMARPOD_STATUS_CODE_UNKNOWN_ERROR`
Returned by function `Smarpod_GetStatusInfo` if it is called with an unknown status code.

`503: SMARPOD_HARDWARE_MODEL_UNKNOWN_ERROR`
Returned by the SmarPod initialization if called with an unknown hardware-model code.

`504: SMARPOD_WRONG_COMM_MODE_ERROR`
Returned by SmarPod initialization if the controller was initialized in synchronous mode.

`505: SMARPOD_NOT_INITIALIZED_ERROR`
Returned by various functions if no SmarPod has been initialized for the given SmarPod ID.

`506: SMARPOD_INVALID_SYSTEM_ID_ERROR`
Returned by `Smarpod_InitSystems` and `Smarpod_Initialize` if an MCS System ID could not be found in the list of connected MCS devices.

`507: SMARPOD_NOT_ENOUGH_CHANNELS_ERROR`
Returned by `Smarpod_Initialize` if the controller specified by the System ID has less channels than required for SmarPods, i.e. if the number of channels is less than 6.

`510: SMARPOD_SENSORS_DISABLED_ERROR`
Returned by commands that move the SmarPod if the sensor mode is *disabled*.

`511: SMARPOD_WRONG_SENSOR_TYPE_ERROR`
: No longer used. Replaced by `SMARPOD_SYSTEM_CONFIGURATION_ERROR`.

`512: SMARPOD_SYSTEM_CONFIGURATION_ERROR`
: Returned by function `Smarpod_Initialize` if the internal MCS configuration does not match the SmarPod model passed to the initialization function. See 3.1 “System Configuration Error” and `Smarpod_ConfigureSystem`.

`513: SMARPOD_SENSOR_NOT_FOUND_ERROR`
: Returned by function `Smarpod_Initialize` if one or more sensors of the SmarPod positioners could not be detected.

`514: SMARPOD_STOPPED_ERROR`
: Returned by `Smarpod_Calibrate` and `Smarpod_FindReferenceMarks` if `Smarpod_Stop` is called while they are executing. Returned by `Smarpod_Move` if called with *waitForCompletion* = 1.

`515: SMARPOD_BUSY_ERROR`
: Returned by functions `Smarpod_Calibrate`, `Smarpod_FindReferenceMarks`, `Smarpod_Move` and `Smarpod_StopAndHold` if the SmarPod is busy and the function cannot be executed. E.g. when the SmarPod is referencing or calibrating, a call of `Smarpod_Move` would return with `SMARPOD_BUSY_ERROR`. `Smarpod_Move` will also returns with `SMARPOD_BUSY_ERROR` if another move command that has been called with *waitForCompletion*=1 is still executing.

`550: SMARPOD_NOT_REFERENCED_ERROR`
: Returned by `Smarpod_Move` if the reference marks of the positioners are not known. See section 2.4.5 “Finding Reference Marks“.

`551: SMARPOD_POSE_UNREACHABLE_ERROR`
: Returned by `Smarpod_Move` if the pose cannot be reached.

`552: SMARPOD_COMMAND_OVERRIDDEN_ERROR`
: When the software commands a movement which is then interrupted by the Hand Control Module, an error of this type is generated.

`553: SMARPOD_ENDSTOP_REACHED_ERROR`
: This error is returned if the target pose could not be reached because a mechanical end stop was detected.

`554: SMARPOD_NOT_STOPPED_ERROR`
: Returned if a command is called which requires that the SmarPod is stopped but it is moving or holding.

`555: SMARPOD_COULD_NOT_REFERENCE_ERROR`
: This error is returned by `Smarpod_FindReferenceMarks` if the reference marks could not be found.