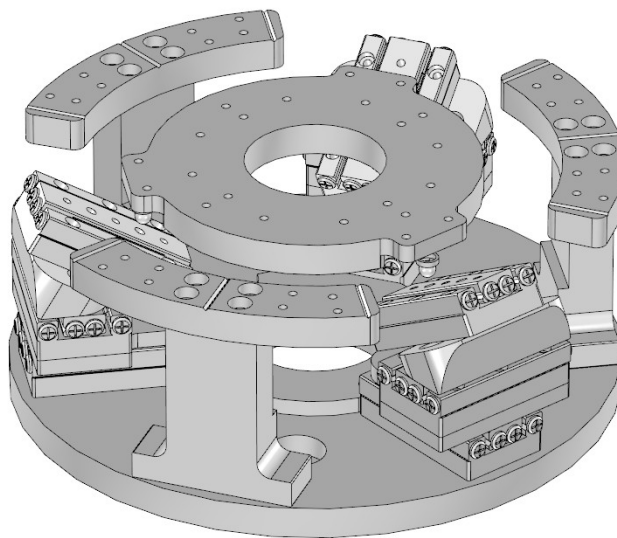


## ***SmarPod Programmer's Guide***



Document Version: 10-11-17-1732  
DLL Version: 1.0.1

Flötenstraße 70  
D 26125 Oldenburg  
<http://www.smaract.de>

Tel.: +49 (0) 441 - 800879 - 0  
Fax: +49 (0) 441 - 800879 - 21  
E-Mail: [info@smaract.de](mailto:info@smaract.de)

# Table of Contents

1 Introduction.....	3
2 Use of SmarPod.....	4
2.1 Technical Overview.....	4
2.1.1 Sensor Modes.....	5
2.2 Initialization.....	6
2.2.1 Initializing MCS Controller Systems.....	6
2.2.2 Initializing the SmarPod.....	6
2.2.3 Calibrating Sensors.....	6
2.2.4 Finding Reference-Marks.....	7
2.3 Using SmarPods and Other Positioners.....	8
2.4 Moving the SmarPod.....	9
2.4.1 Pose.....	9
2.4.2 Movement Velocity.....	9
3 Function Reference.....	11
Smarpod_GetDIIVersion.....	11
Smarpod_GetStatusInfo.....	12
Smarpod_InitSystems.....	13
Smarpod_ReleaseAll.....	14
Smarpod_Initialize.....	15
Smarpod_Calibrate.....	16
Smarpod_FindReferenceMarks.....	17
Smarpod_IsReferenced.....	18
Smarpod_SetSensorMode.....	19
Smarpod_GetSensorMode.....	20
Smarpod_SetMaxFrequency.....	21
Smarpod_GetMaxFrequency.....	22
Smarpod_SetSpeed.....	23
Smarpod_GetSpeed.....	24
Smarpod_SetPivot.....	25
Smarpod_GetPivot.....	26
Smarpod_IsPoseReachable.....	27
Smarpod_Move.....	28
Smarpod_Stop.....	29
4 Appendix.....	30
4.1 Code Example.....	30
4.2 Status Codes.....	31

# 1 Introduction

The *SmarPod Programmer's Guide* describes the programming of *SmarAct's SmarPod* manipulators. The *SmarPod Application Programming Interface (API)* and the *SmarPod DLL* (file *SmarPod.dll*) are required to write software that initializes, configures and moves SmarPods.

The SmarPod Installer installs the following files into the SmarAct folder:

- the header file *SmarPod.h* contains function declarations, types and constants
- the library *SmarPod.lib*
- the DLL *SmarPod.dll* which is loaded by the program that was linked against *SmarPod.lib*
- the DLL *MCSCControl.dll* which is required by *SmarPod.dll*
- several other DLLs also required by *SmarPod.dll*
- code examples
- this documentation and additional documents.

The SmarAct folder is located in the programs folder on the Windows system partition if the user has not chosen a different location at installation.

All DLLs required for SmarPod control must be visible for the software when it is launched. The program can load a DLL if it is in the same directory as the program's executable file or if the DLL is installed in the Windows system directory. If your software is installed on other computers the DLLs must be installed as well.

While the *Modular Control System (MCS)* DLL *MCSCControl.dll* must be installed to control a SmarPod, the MCS C headers and library are not needed for programming the SmarPod. However, both programming interfaces can be used if a program needs to control other MCS driven positioners beside SmarPods. For a discussion of this case see section 2.3 "Using SmarPods and Other Positioners".

For a technical overview of the hardware see section 2.1 "Technical Overview".

## 2 Use of SmarPod

### 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.

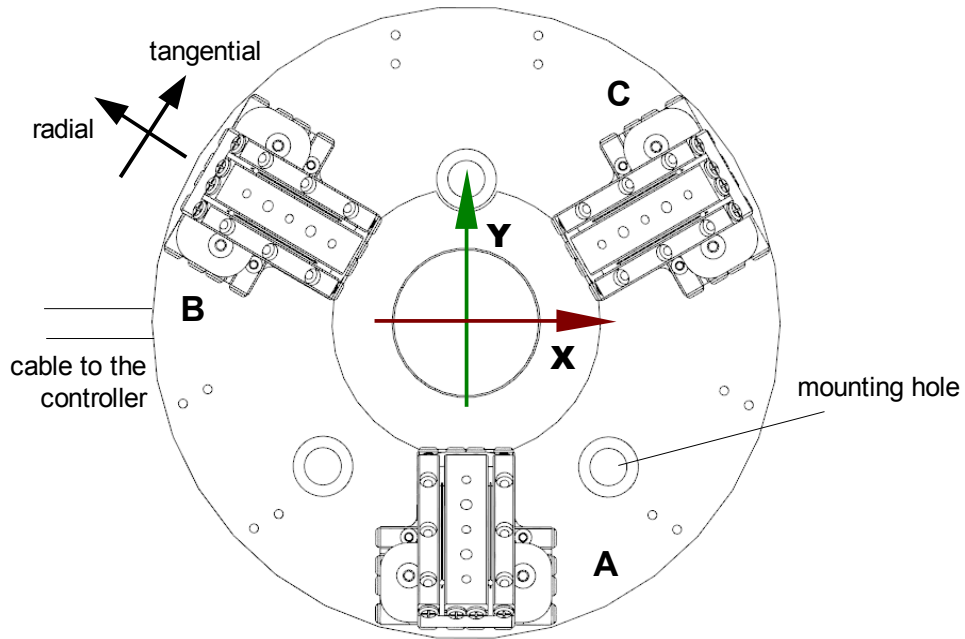


Figure 1: A SmarPod (without the top-plate) from above

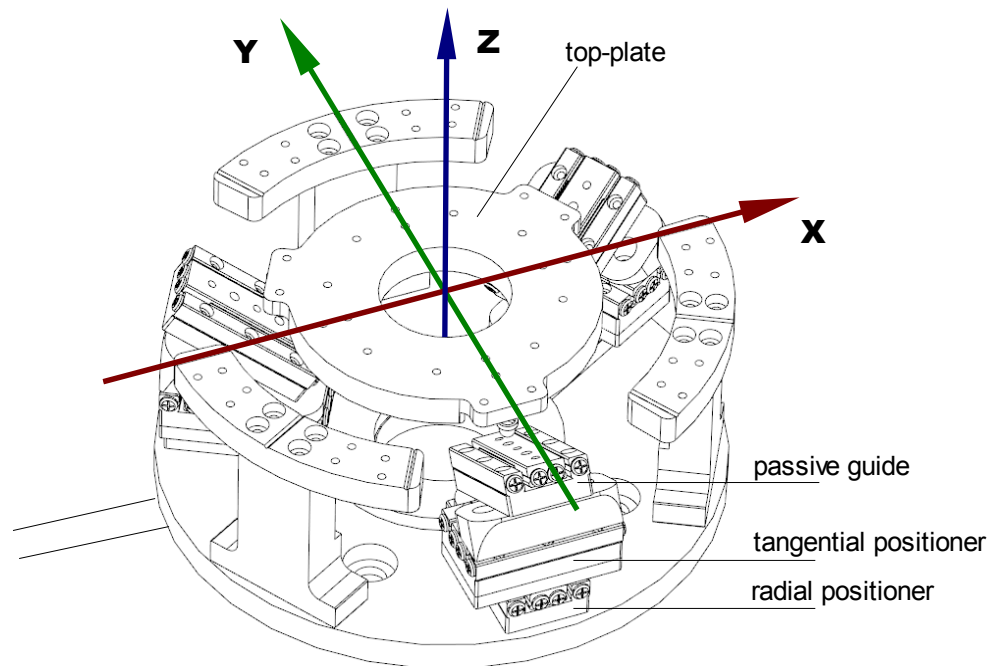


Figure 2: The coordinate system axes of a SmarPod

### 2.1.1 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.

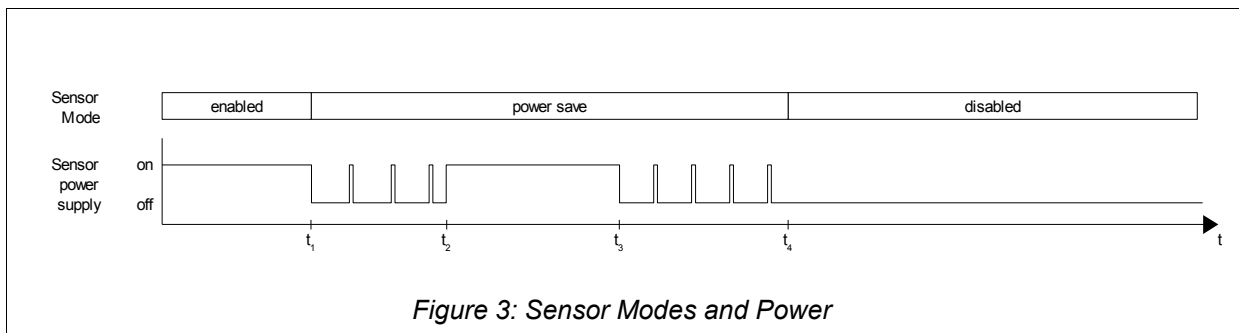


Figure 3: Sensor Modes and Power

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.

## 2.2 Initialization

A SmarPod must be initialized before it can be used. Initialization tells the DLL which SmarPod hardware model is connected and by which MCS it is controlled. The initialization sequence:

1. initializing the controller(s) (see 2.2.1 Initializing MCS Controller Systems)
2. initializing the SmarPod (see 2.2.2 Initializing the SmarPod).
3. activating the sensors (see 2.1.1 Sensor Modes).
4. calibrating the SmarPod positioners (rarely necessary, see 2.2.3 Calibrating Sensors).
5. finding the positioner's reference-marks (see 2.2.4 Finding Reference-Marks).

### 2.2.1 Initializing MCS Controller Systems

Before a SmarPod can be used by a program its controller must be initialized. When calling the command `Smarpod_InitSystems` all MCS controllers to be initialized must be specified by a list containing the controller's *System IDs*. System IDs are part of the device's serial numbers which can be found on the controller cases. They have the format:

```
1547500793.150
```

where the first part is the System ID (here: 1547500793).

All controllers which the program uses must be initialized at the same time. It is not possible to initialize some controllers at one point and initialize additional controllers later. Before other controllers can be initialized `Smarpod_ReleaseAll` must be called.

If only one controller is connected to the host PC there is an easier method. If the list of System IDs contains only one element with value 0, the first controller found in the list of MCS USB devices is taken.

Using 0 instead of a real MCS System ID is not recommended if more than one controller is connected, because the order and number of devices may change between sessions.

### 2.2.2 Initializing the SmarPod

When the controller is initialized, `Smarpod_Initialize` must be called for every SmarPod you intend to use. The function specifies the model of the SmarPod product family and the controller to which it is connected.

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

The DLL supports controlling of (`SMARPOD_MAX_ID+1`) SmarPods in one program (currently 8). A *SmarPod ID* must be specified when calling `Smarpod_Initialize` which will be used to address the SmarPod in function calls later. The ID can be freely chosen, as long as it is not already in use and it is between 0 (included) and the constant `SMARPOD_MAX_ID` from the header file (currently 7, also included).

The *System ID* parameter tells the DLL which MCS controller the SmarPod is connected to. If you want to select the first MCS seen by the DLL pass a value of 0.

### 2.2.3 Calibrating Sensors

When a SmarPod is connected to a controller for the first time, the sensors of the positioners must be calibrated. For SmarPods that come preconfigured from SmarAct, this is usually not necessary. The calibration data is stored in non-volatile memory of the controller and does not have to be repeated if the controller is switched off.

During calibration each SmarPod positioner is moved by short distances for a few seconds. The calibration must not be performed while the positioners are near mechanical end stops. Also please remove any load from the SmarPod so the positioners can move easily.

The function `Smarpod_Calibrate` can be used for calibrating. Calibration can also be done using the programs *MCSCalibrator* or *MCSConfiguration*, if available.

## 2.2.4 Finding Reference-Marks

After controller and SmarPod have been initialized and before the SmarPod can be used, the controller needs to find the reference-marks of all positioners to read their absolute positions. This is done by the function `Smarpod_FindReferenceMarks`. The controller remembers the reference data until the next reset. It is not necessary to perform reference-finding at every start of a program. Only if the *referenced*-state returned by `Smarpod_IsReferenced` is false. Since a controller loses the reference data after it is switching off, reference-finding must be performed at least once after switching on. When `Smarpod_FindReferenceMarks` has finished the SmarPod is at its zero position.

`Smarpod_FindReferenceMarks` moves all SmarPod positioners with high velocity over their full travel range. Before calling the function please ensure that this does not cause collisions.

## 2.3 Using SmarPods and Other Positioners

*This section is for programmers who want to write software for SmarPods and other SmarAct positioners controlled by MCS controllers. Please consult the MCS Programmer's Guide for a documentation of the MCS functions mentioned here.*

The SmarPod API can be used independently from the SmarAct *Modular Control System* (MCS) API. To write software that controls only SmarPods, only the SmarPod API is needed. It is, however, possible to use both APIs to control SmarPods and other positioners by one application. In this case there are several points that must be considered:

- `Smarpod_InitSystems` or `SA_InitSystems` (MCS) can be used to initialize the controllers. One difference between the two functions is the specification of controller systems. `Smarpod_InitSystems` expects an array of System IDs, while `SA_InitSystems` uses an internal list, which can be cleared and filled with System IDs.
- Since the SmarPod DLL must communicate in asynchronous communication mode with SmarPod controllers and because the communication mode must be the same for all controllers, `SA_InitSystems`, if used for the initialization, must be called with the mode selector `SA_ASYNCHRONOUS_COMMUNICATION`. If `SA_InitSystems` initializes the synchronous mode, calls to `Smarpod_Initialize` fail with a wrong-mode error. `Smarpod_InitSystems` automatically sets the async mode.
- As a consequence the program must communicate with all positioners using asynchronous commands which demands a software control-flow that can handle asynchronous messaging. This should be considered early in the development of the software.
- With the MCS function `SA_SetReceiveNotification_A` a *Windows event* can be set per controller to get notifications about data packets received from the controller. The SmarPod DLL needs to set this event exclusively for SmarPod controllers. Therefore user code must not set an event for a SmarPod controller. Setting events for other controllers is OK.
- Although it is possible to send commands to SmarPod controllers using MCS functions, this would interfere with the SmarPod DLL state, which leads to undefined behavior and is not recommended.

## 2.4 Moving the SmarPod

The top-plate 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.

### 2.4.1 Pose

The translation and orientation of the top-plate is called the *pose* of a SmarPod. The translation part of a pose is the offset of the top-plate 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 locations of interest. Since the pivot-point is usually fixed or changed rarely, it is not a parameter of the move command. Use the function `Smarpod_SetPivot` to change it. After initialization the pivot-point is (0,0,0). With a pivot of (0,0,0) the top-plate rotates around the center of the triangle of the joints between the top-plate and the passive guide axes (see Figure 4).

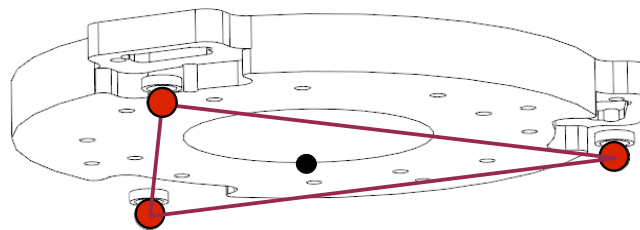


Figure 4: The default pivot-point

Note that the default pivot is below the surface of the top-plate because at this point the range of rotation angles is maximal.

Figure 5 illustrates how the top-plate (box) is rotated by the same angles  $\Theta$  and  $-\Theta$  around different pivot-points (small circle).

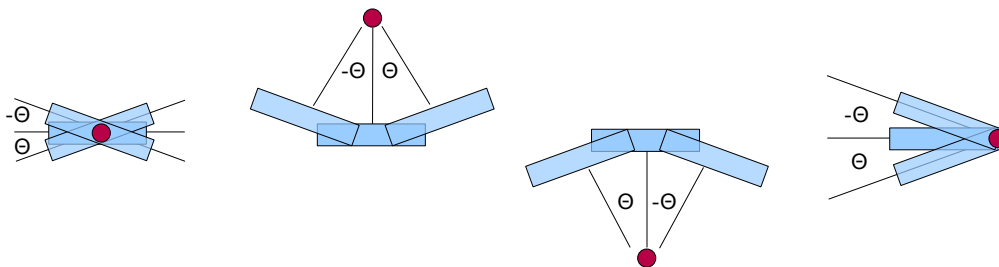


Figure 5: Rotations of the top-plate for different pivot-points

### 2.4.2 Movement Velocity

The software can specify movement velocities with the function `Smarpod_SetSpeed`. The maximum velocity a SmarPod can reach depends on the capabilities of its positioners and can be reduced by its load. If the highest velocity reachable is lower than the specified velocity, the SmarPod will move slower, but it will not 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 are reported as `SMARPOD_ENDSTOP_REACHED_ERROR`. In such cases the SmarPod stops moving and does not reach the target pose. If this happens you should set the speed to a lower value.

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

`Smarpod_SetSpeed` sets the speed-control mode:

- If *speed-control* is disabled the *speed* parameter has no meaning. All positioners are driven with the current maximum frequency (see `Smarpod_SetMaxFrequency`), so they move with approximately the same velocity. In this mode the top-plate does not necessarily move smoothly from the start to the end pose because some positioners may reach their target positions before others. As an effect the top-plate can tilt temporarily during the movement. Generally, movements with disabled speed-control are faster but less predictable than those with speed-control enabled.

Movements with disabled speed-control should be used to move the SmarPod quickly to certain poses if tilting is not critical.

- If *speed-control* is enabled, the top-plate 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 to perform a smooth transition. The *speed* parameter of `Smarpod_SetSpeed` determines the velocity of the fastest moving positioner in meters per second. All other positioners will move slower or with the same velocity.

## 3 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.dll. 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.dll 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 4.2 “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\_InitSystems***

### **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 are initialized and used.

See section 2.2.1 “Initializing MCS Controller Systems” for more information.

See section 2.3 “Using SmarPods and Other Positioners” for informations about initializing controllers using functions from the MCSControl DLL instead of `Smarpod_InitSystems`.

### **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\_ReleaseAll***

### **Interface:**

```
Smarpod_Status Smarpod_ReleaseAll();
```

### **Description:**

This function releases all initialized MCS controllers.

### **Example:**

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

## ***Smarpod\_Initialize***

### **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).

The caller must pass the 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 will succeed but the execution of commands will lead to undefined behavior.

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

See section 2.2 “Initialization” for more information.

### **Parameters:**

- *smarpodId* (unsigned 32 bit), input – an ID for the SmarPod. Can be chosen 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 (already initialized) 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\_Calibrate***

### **Interface:**

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

### **Description:**

This function calibrates all positioners of the SmarPod.

See also section 2.2.3 “Calibrating 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 function blocks the calling thread until all reference-marks are found or an error has occurred.

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.

See also “`Smarpod_IsReferenced`” and section 2.2.4 “Finding Reference-Marks”.

### **Parameters:**

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

### **Example:**

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

## ***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 result;  
int referenced;  
result = Smarpod_IsReferenced(smarpodId, &referenced);  
if(result == SMARPOD_OK) {  
    if(referenced == 0)  
    {  
        result = 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 2.1.1 “Sensor Modes” for more information.

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

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

### **Parameters:**

- *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 2.1.1 “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 2.4 “Moving the SmarPod” for more information about the relationship between maximum frequencies 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 <i>speed</i> parameter has no effect here. The actual speed depends on the maximum frequency which can be set with <code>Smarpod_SetMaxFrequency</code> . 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 <i>speed</i> parameter determines the speed of the fastest moving positioner.

See section 2.4 “Moving the SmarPod” for more informations 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 2.4 “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\_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 modify the SmarPod's state. 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.
- *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\_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 pose can be reached. If it cannot be reached, the function fails with `SMARPOD_POSE_UNREACHABLE_ERROR`.

See also section 2 “Use of SmarPod”.

### **Parameters:**

- *smarpodId* (unsigned 32 bit), input – The ID of the SmarPod.
- *pose* (pointer to struct `Smarpod_Pose`), input – The pose to move to.
- *holdTime* (unsigned 32 bit), input – Specifies how long (in milliseconds) the pose is actively held after reaching the target. The valid range is 0..60,000. A 0 deactivates this feature. Pass `SMARPOD_HOLDTIME_INFINITE` (60,000) 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.

See also section “Smarpod\_Move”.

### **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);
}
```

## 4 Appendix

### 4.1 Code Example

A simple code example demonstrating initialization, use and release of a SmarPod.

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

const int kCalibrate = 0; /* set to 1 to calibrate */
const unsigned int kSmarpodId = 0;
const unsigned int kSmarpodModel = 10001; /* model = SmarPod 110.45 S */
const unsigned int kMcsId = 0; /* use the first MCS controller */

const double kXyMax = 0.0102;
const Smarpod_Pose pZero = { 0.0,0.0,0.0, 0.0,0.0,0.0 };
const Smarpod_Pose pZ0Left = { -kXyMax,0.0,0.0, 0.0,0.0,0.0 };
const Smarpod_Pose pZ0Right = { kXyMax,0.0,0.0, 0.0,0.0,0.0 };

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

    Smarpod_GetDLLVersion(&major,&minor,&update);
    printf("loaded SmarPod.dll version: %u.%u.%u\n",major,minor,update);
    printf("compiled with SmarPod API version: %u.%u.%u\n",
        SMARPOD_API_VERSION_MAJOR, SMARPOD_API_VERSION_MINOR, SMARPOD_API_VERSION_UPDATE);

    initList[0] = kMcsId;
    status = Smarpod_InitSystems(initList,1);
    if(status)
        return 1;
    status = Smarpod_Initialize(kSmarpodId,kSmarpodModel,kMcsId);
    if(status)
        return 1;

    Smarpod_SetSensorMode(kSmarpodId,SMARPOD_SENSORS_ENABLED);
    if(kCalibrate)
        status = Smarpod_Calibrate(kSmarpodId);
    if(status)
        return 1;

    Smarpod_IsReferenced(kSmarpodId,&referenced);
    if(!referenced)
        status = Smarpod_FindReferenceMarks(kSmarpodId);
    else
        status = Smarpod_Move(kSmarpodId,&pZero,SMARPOD_HOLDTIME_INFINITE,1);
    if(status)
        return 1;

    Smarpod_SetSpeed(kSmarpodId,1,0.002);
    Smarpod_Move(kSmarpodId,&pZ0Left,SMARPOD_HOLDTIME_INFINITE,1);
    Smarpod_SetSpeed(kSmarpodId,1,0.006);
    Smarpod_Move(kSmarpodId,&pZ0Right,SMARPOD_HOLDTIME_INFINITE,1);
    Smarpod_SetSpeed(kSmarpodId,1,0.01);
    Smarpod_Move(kSmarpodId,&pZero,SMARPOD_HOLDTIME_INFINITE,1);

    Smarpod_ReleaseAll();
    return 0;
}
```

## 4.2 Status Codes

SMARPOD\_OK

The function call was successful.

SMARPOD\_OTHER\_ERROR

An uncategorized error occurred.

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.

SMARPOD\_NO\_SYSTEMS\_FOUND\_ERROR

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

SMARPOD\_INVALID\_PARAMETER\_ERROR

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

SMARPOD\_COMMUNICATION\_ERROR

Can be returned by various functions if there is a communication problem with the MCS controller.

SMARPOD\_STATUS\_CODE\_UNKNOWN\_ERROR

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

SMARPOD\_HARDWARE\_MODEL\_UNKNOWN\_ERROR

Returned by the SmarPod initialization if called with an unknown hardware-model code.

SMARPOD\_WRONG\_COMM\_MODE\_ERROR

Returned by SmarPod initialization if the controller was initialized in synchronous mode.

SMARPOD\_NOT\_INITIALIZED\_ERROR

Returned by various functions if no SmarPod has been initialized for the given SmarPod ID.

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.

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.

SMARPOD\_SENSORS\_DISABLED\_ERROR

Returned by commands that move the SmarPod if the sensor mode is *disabled*.

SMARPOD\_NOT\_REFERENCED\_ERROR

Returned by `Smarpod_Move` if the reference-marks of the positioners are not known. See section 2.2.4 "Finding Reference-Marks".

SMARPOD\_POSE\_UNREACHABLE\_ERROR

Returned by `Smarpod_Move` if the pose cannot be reached.

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.

SMARPOD\_ENDSTOP\_REACHED\_ERROR

This error is generated if the target pose could not be reached because a mechanical end stop was detected.