SmarActMC
Motion Control API
Programmer's Guide

– Pre-Release! –

www.smaract.com

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

Document Version 0.11.0.2, Build 34

TABLE OF CONTENTS

1 INTRODUCTION

SmarAct Motion Control (*SmarActMC* for short) is an application programming interface (*API*) for different SmarAct motion devices.

This document describes the usage of the API. In the first chapters, the general concepts are described while the later chapters contain overviews of API functions, data structures and constants.

1.1 Pre-Release Status

The document is a ***pre-release version****.*
It is incomplete and probably contains inaccurate information!

If you intend to write software based on a pre-release version of SmarActMC (version numbers < 1.0), please keep the following in mind:

- Values and names of constants may change. This includes error codes, property keys, and property values.
- New properties and functions may be added, renamed or removed.
- The software is likely to contain bugs which can result in erroneous behavior of the positioning device even if the API is used correctly. It is advisable to test your application under conditions where an unwanted movement of the positioning device cannot do any harm.

1.2 Programming Language Support

The SmarActMC API is included in the software development kits (SDK) for different programming languages:

- **SmarActMC SDK for C/C++**: the SmarActMC library can be linked into an application as a shared library. Functions and constants are provided as a C header file.
  The C API functions and constants are prefixed with *SA_MC_...*.
- **SmarActMC SDK for Python 3**: the API is provided by the Python package *smaract.mc*.

The API reference in this document is generally programming language-independent but refers to individual languages where necessary. Function and constant names are generally written without the "SA_MC_" prefix except for code examples in C. When using the C API this prefix has to be prepended to the name.

1.3 Installation of the SDK

C/C++ SDK

The Linux SDK consists of several shared libraries: *libsmaractmc.so* library contains the application programming interface (API). The SDK also contains the supporting libraries *libsmaractctl.so*, *libsmaractio.so* and symbolic links to these libraries.

The libraries have been compiled with the g++ 4.9 compiler. The run-time libraries for that compiler must be available on the host system.

The C header *SmarActMC.h* contains the function declarations and constants of the API.

We assume that you know how to install libraries under Linux such that they can be loaded by your application (but feel free to contact SmarAct for help).

A C++ code example should be included which demonstrates the basic steps to initialize and move a device.

Python SDK

Because Python APIs for SmarAct products are very similar in design and behavior, usage instructions and their similarities and differences to the C API are described in the document *Using SmarAct Python SDKs.pdf* which is included in the software bundle.

Python programming examples are included if the Python SDK is installed. They are a good starting point for writing software in Python.

2 GENERAL CONCEPTS

2.1 Positioning Devices

A SmarAct positioning device supported by the SmarActMC software belongs to a *device class*. The device class defines the basic mechanical structure of the positioner. For example, the *TriPod* product series is characterized by an endeffector-interface (the top-plate) which is deflected by three linear axes. The *TriPod* series contains several product lines on same structure but with different properties like size, travel range and the supported number of degrees of freedom.

In order to communicate with the device, the motion software needs to know its *software model code*. The model code is an important parameter that must be passed to the `Open` function when initializing the device, see 2.4.

The software model code should be included in the documentation that was shipped with the device. It can be found, for example, in the Positioning System Test Report.

Please make sure that you use the right model code. Otherwise the device might behave unexpectedly and could be damaged or damage other objects.

2.2 Error Handling

If a function fails during its execution, it terminates with an error. Errors are distinguishable by the *error code*. A list of the error codes can be found in section 8.

Always implement a proper error handling! If errors are ignored by your software, the device may behave unexpectedly and could be damaged or damage other objects.

C/C++

All C API functions return a result value which indicates either no error (*SA_MC_ERROR_NONE*) or an error (*SA_MC_ERROR_...*). When using the C API (see 1.2) it is recommended to check the result returned by each function and handle errors. For example:

```
SA_MC_Result result = SA_MC_ERROR_NONE;
int32_t holdTime = 0;
result = SA_MC_GetProperty_i32(hDevice, SA_MC_PKEY_HOLD_TIME, &holdTime);
if(result != SA_MC_ERROR_NONE) {
    // handle the error
} else {
    // property value read successfully
}
```

Python

The Python API functions raise exceptions of type `smaract.mc.Error` in cases of errors but do not return error codes like the C API functions. The exception contains the error code, see 4.2.1.

```
import smaract.mc as mc
try:
    # ...
    holdTime = mc.GetProperty_i32(hDevice, mc.Property.HOLD_TIME)
    # if we are here, the property value has been read successfully.
    # if GetProperty_i32 has raised an exception, we will not get here...

except mc.Error as err:
    # a MC function within the try block has raised an exception.
    # print the function name and the error code:
    print("SmarActMC error: %s -> %s (%s) " %
          (err.func, mc.ErrorCode(err.code), err.code))
```

2.3 Controller Locators

A *controller locator* identifies a SmarAct MCS2 controller. The locator is a text string that contains the communication interface and the address of the controller. The locator must be passed in the options parameter of the `Open` function, see 2.4.

2.3.1 Locator Formats

The MCS2 supports different communication interface types. The locator format for each interface is slightly different.

Locators for the USB interface

The general USB locator format is:

`usb:<address>`

Options for <*address*> are:	Examples:
`usb:sn:<serialnumber>`	`usb:sn:MCS2-00001234`
`usb:ix:<index>`	`usb:ix:0`

<*serialnumber*> is the serial number of an MCS2 controller.

<*index*> selects the *n*th MCS2 controller (starting with 0) from the list of all currently connected controllers USB ports on the host computer. The drawback of identifying a controller with this method is, that the number and order of the connected controllers can change, so the index may not always refer to the same controller. It is only safe to use this address format if exactly one controller is connected to the computer.

We therefore recommend to use the `usb:sn:<serialnumber>` format.

Locators for the network interface

The general network locator format is:

`network:<address>`

Options for the *<address>* part are:	Examples:
`network:<ipv4>`	`network:192.168.1.200`
`network:sn:<serialnumber>`	`network:sn:MCS2-00001234`

<ipv4> is an IPv4 address which consists of four integer numbers between 0 and 255 separated by a dot.

The locator `network:sn:MCS2-00001234` identifies MCS2 with the serial number MCS2-00001234 on the network.

*Do not add leading zeros to the numbers in the IPv4 address as these would be interpreted as octal numbers. The IPv4 addresses 192.168.050.200 and 192.168.50.200 are **not** the same!*

2.3.2 Searching for Controllers

Function FindControllers can be used to search for MCS2 controllers connected via USB or ethernet. The function returns a string with locators, separated by newline characters.

C/C++

```
char buffer[1024];
size_t bufferSize = sizeof(buffer);
result = SA_MC_FindControllers("",buffer,&bufferSize);
if(result == SA_MC_ERROR_NONE) {
  /* buffer contains a list of locators, separated by newlines, or is an empty
string */
}
```

Python

```
locators = smaract.mc.FindControllers()
# locators is a string with a list of controllers, separated by newlines, or is
empty
```

2.4 Device Initialization

A device must be initialized with the `Open` function before it can be used. Only one application can control a device at a time. If a second one tries to initialize the same device, `Open` will return with an error code.

Some steps must be carried out before an application can start moving the device:

- Locate the controller on the USB bus or the network. If the address of the controller is not known, the function `FindControllers` can be used to search for controllers. See 2.3.2.
- Call the Open function (5.1.2). Open needs the locator (2.3) and the model code (2.1) of the positioning device as options, separated by a newline character, for example: "locator network:sn:MCS2-00001234\nmodel 22000" If successful,

Open returns a *handle* which identifies the device. The handle must be passed to subsequent calls to API functions. If multiple devices are used by the program simultaneously, each one has a different handle.

- If the device has not been referenced since the last power-cycle of the controller, it must be referenced, see 2.5.
- When the application has finished using the device, call `Close` to close the device and to disconnect the software from the controller. After closing, the handle is no longer usable.

When calling `Open` the model and locator options must be correct for your device and controller. Otherwise, using the device can cause unwanted movements and could damage the device or other objects.

2.5 Referencing

Referencing (also known as homing) is the process of determining the absolute position of an axis.

All axes must be referenced before a device can be used. This has to be done only once after powering up a SmarAct controller. If they are not referenced, the `Move` function will return with *ERROR_NOT_REFERENCED*. The referencing status of the device can be checked by reading the *IS_REFERENCED* property, see 2.7.

Start a referencing operation by calling the `Reference` function. The program can wait for the termination of `Reference` by waiting for a *MOVEMENT_FINISHED* event with `WaitForEvent`. If the referencing procedure fails, an error is returned (C/C++) or an exception is raised (Python).

Referencing can involve several phases and the axes may change their movement direction several times.

Ensure that the device has sufficient travel space to avoid collisions when referencing or the device might get damaged or damage other objects!

The actual movements are influenced by the physical positions of the axes before starting to reference. SmarAct positioners reference by searching for *reference marks* on the sensor strip. The movment of an axis, when referencing, depends on how far its sensor is from the next reference mark and wether the mark lies before or behind the sensor, relative to the direction of movement.

Don't assume that every referencing operation of a device performs the exact same movements of the device axes.

2.6 End Effector Pose

The target position of a movement is defined by the *pose* parameter of the `Move` function. A *pose* is a hex-tuple (*x, y, z, rx, ry, rz*) that describes the translation and the orientation of the end effector. *x*, *y* and *z* are the three translation offsets in meters, and *rx*, *ry* and *rz* the rotation angles in degrees.

A device might not support all six degrees of freedom. For unsupported degrees of freedom, the unsupported pose parameters must be set to 0 when calling the `Move` function.

For example, a *TiltStage* device does not support rotation around the *Y*-axis. Thus, the *RY* posetg parameter must be set to 0 when calling the *Move* command or it will return with an error.

If called with a pose that is not physically reachable or not allowed, `Move` will return with *ERROR_POSE_UNREACHABLE.*

2.7 Properties

Properties are values that can be accessed by a unique code, the *property key*. Some properties can be read and written, some are read-only. Read-only properties indicate a state, like *IS_REFERENCED* or information about the device, like *MODEL_NAME*. Writable properties can be used to change the behavior of a device, for example, *HOLD_TIME*.

The property key must be passed to the GetProperty... and SetProperty... functions (5.5.1) to select the property.

Properties have a data type: *int32*, *double* (f64) or *string*. Use the get or set function that matches the data type: `Set/GetProperty_i32` for *int32*, `Set/GetProperty_f64` for double floating point values and `GetProperty_s` for string properties. The data types for each property can be found in section 7.

For example, the property *HOLD_TIME* has the property key *0x2000* (in hexadecimal format) and is of type *int32*, so it must be written with `SetProperty_i32` and read with `GetProperty_i32`. If you want to change the hold time to 500 milliseconds, write:

C/C++

```
result = SA_MC_SetProperty_i32(handle, SA_MC_PKEY_HOLD_TIME, 500);
```

Python

```
smaract.mc.SetProperty_i32(handle, smaract.mc.Property.HOLD_TIME, 500);
```

2.8 Event Notifications

An *event notification* is created when an event occurs. For example, when a movement operation has finished successfully or failed, a *MOVEMENT_FINISHED* event notification is created.

Event notifications can be waited for with the `WaitForEvent` function (5.2.1). A call to `WaitForEvent` blocks the calling thread until:

- an event is available,
- the maximum time for waiting, specified in the *timeout* parameter, has elapsed,
- the `Cancel` function was called from a different thread,
- an unexpected error has occurred, for example, the communication with the hardware controller was interrupted.

The event data returned by `WaitForEvent` contains details about the event, including the *event type*. The meaning of the fields of the structure depend on the event type, see 4.1.4.

The event data is valid only if `WaitForEvent` returns with no error.

`WaitForEvent` waits until any event is available, not only *MOVEMENT_FINISHED*. Therefore, the calling code must check the event type and handle each type appropriately. If your code should block until the end of a motion operation, it should call `WaitForEvent` in a loop until it receives the expected *MOVEMENT_FINISHED* event.

With a timeout of *INFINITE* the function will block forever if no event becomes available, no error occurs and `Cancel` is not called.

Examples

C/C++

```
SA_MC_Event ev;
SA_MC_Result res = SA_MC_WaitForEvent(handle,&ev,5000); // timeout is 5 seconds

if(res == SA_MC_ERROR_NONE) {
  switch(ev.type) {
  case SA_MC_EVENT_MOVEMENT_FINISHED:
    if(ev.i32 == SA_MC_ERROR_NONE) {
      std::cout << "movement finished" << std::endl;
    } else {
      std::cout << "movement finished with status: " << ev.i32 << std::endl;
    }
    break;
  default:
    std::cout << "unhandled event received" << std::endl;
    break;
  }
} else if (res == SA_MC_ERROR_TIMEOUT) {
  std::cout << "waiting for event notification timed out" << std::endl;
} else if (res == SA_MC_ERROR_CANCELED) {
  std::cout << "waiting for event notification was canceled" << std::endl;
} else {
  std::cout << "error while waiting: " << res << std::endl;
}
```

Python

```
import smaract.mc as mc

try:
  ev = mc.WaitForEvent(handle,5000)     # timeout is 5 seconds

  if ev.type == mc.EventType.MOVEMENT_FINISHED:
    if ev.i32 == mc.ErrorCode.NONE:
       print("movement finished")
    else:
       print("movement finished with status %s" % mc.ErrorCode(ev.i32))
  else:
    print("unhandled event received")

except mc.Error as err:
  # handle errors including timeout
```

```
    print("WaitForEvent raised an exception: %s" % mc.ErrorCode(err.code))
```

2.9 Asynchronous Operations

Functions that initiate a motion operation are *asynchronous*. This means, the function returns immediately – not when the operation itself has finished! If the call to `Move` or `Reference` returns with an error, this indicates that the operation could not be started.

When the movement finishes – successfully or not – a *MOVEMENT_FINISHED* event notification is generated, see 2.8.

3 MOVING

Function Move starts a movement of the device to a specfied target pose. With Stop a movement can be interruped.

3.1 Movement Speed

There are two properties to set the speed of the end effector: *MAX_SPEED_LINEAR_AXES* (in meters per second) and *MAX_SPEED_ROTARY_AXES* (in degrees per second). The values determine the maximum speed of the respective class of axes (linear or rotary) for the next movement:
When a movement command is executed, the maximum speed specified in these properties is assigned to the axis that will move the greatest distance. The other axes, which travel smaller distances, will move with a speed that is proportional to the ratio of the largest travel distance and their own travel distance.
For example, if *MAX_SPEED_LINEAR_AXES* is set to *2 mm/s* and three linear axes have to travel distances of 1 mm, 0.5 mm and 0.1 mm respectively to reach the target pose, the first one will move with the maximum speed *s*, the second one with *s* x 0.5 and the third one with *s* x 0.1, i.e. the speeds are:
2 mm/s, 1 mm/s and 0.2 mm/s.

3.2 Waiting for Finished or Failed Movements

The application can wait until the movement has finished by calling WaitForEvent. A *MOVEMENT_FINISHED* event notification is returned when the last movement has either completed successfully, was stopped with Stop or failed.

A movement may fail during execution. This can happen when axes are facing a large counter force or when they collide with physical obstacles including their own end-stops. A movement failure can be identified by waiting for the *MOVEMENT_FINISHED* event (see 2.8) and checking the *i32* field of the returned event which contains the operation error status like *ENDSTOP_REACHED* or *FOLLOWING_ERROR_LIMIT_REACHED*.

C/C++

```
SA_MC_Pose = { 0.,0.,0.,0.,0.,0. };
SA_MC_Result res = SA_MC_Move(handle, &pose);
if(res == SA_MC_ERROR_NONE) {
  return;
}

SA_MC_Event ev;
SA_MC_Result res = SA_MC_WaitForEvent(handle,&ev,5000); // timeout is 5 seconds

if(res == SA_MC_ERROR_NONE) {
  switch(ev.type) {
  case SA_MC_EVENT_MOVEMENT_FINISHED:
    if(ev.i32 == SA_MC_ERROR_NONE) {
      std::cout << "movement finished" << std::endl;
    } else {
      std::cout << "movement finished with status: " << ev.i32 << std::endl;
    }
    break;
  default:
    std::cout << "unhandled event received" << std::endl;
    break;
  }
} else {
  std::cout << "error while waiting: " << res << std::endl;
}
```

Python

```
import smaract.mc as mc

try:
  mc.Move(handle, Pose(0,0,0,0,0,0) )

  ev = mc.WaitForEvent(handle,5000)     # timeout is 5 seconds

  if ev.type == mc.EventType.MOVEMENT_FINISHED:
    if ev.i32 == mc.ErrorCode.NONE:
       print("movement finished")
    else:
       print("movement finished with status %s" % mc.ErrorCode(ev.i32))
  else:
    print("unhandled event received")

except mc.Error as err:
  # handle errors including timeout
  print("Caught smaract.mc exception: %s" % mc.ErrorCode(err.code))
```

4 DATA TYPE REFERENCES

4.1 Common Data Types

4.1.1 Result

The result type is the return type of C/C++ API functions (*SA_MC_Result*) which return with one of the *SA_MC_ERROR_...* codes. In Python, result values are fields in exception objects which are raised if a function execution fails, see 4.2.1.

4.1.2 Handle

A handle is used to select an object when calling an API function with a handle parameter. Handles are generated by the `Open` function, see 2.4.

4.1.3 PropertyKey

A property key is a code that selects a property, see 2.7.

4.1.4 Event

The *Event* data structure contains the data of one event notification, see 2.8. The *type* field describes the event type. Its value can be one of the event type constants in 6. The meaning of the remaining fields depend on the event type.

C/C++

```
typedef struct SA_MC_Event {
    uint32_t type;
    uint8_t unused[28];
    union {
        int32_t i32;
        int64_t i64;
        uint8_t reserved[32];
    };
} SA_MC_Event;
```

Python

smaract.mc.Event is a class with the attributes

```
Event.type
Event.i32
Event.i64
```

Currently the only fields that are defined in Event beside *type* are

- *i32*: a signed integer 32 bit wide number.
- *i64*: a signed integer 64 bit wide number.

Which field contains additional event information depends on the event type.

4.1.5 Pose

A *Pose* is a hex-tuple for describing end effector poses which are passed as a parameter to the `Move` function, see 2.6.

It contains three offset values *x*, *y* and *z* in meters and three angles *rx*, *ry* and *rz* in degrees.

C/C++

```
typedef struct SA_MC_Pose {
    double x;
    double y;
    double z;
    double rx;
    double ry;
    double rz;
} SA_MC_Pose;
```

Python

smaract.mc.Pose is a Python class with the attributes

```
Pose.x
Pose.y
Pose.z
Pose.rx
Pose.ry
Pose.rz
```

4.2 Types in the Python API

The Python API has some data types which are not available in the APIs for other languages.

4.2.1 Error Exception

The `smaract.mc.Error` exception is raised when the execution of a function fails. For a discussion of handling errors, please refer to section 2.2. The exception contains:

- `Error.code`: the error result code (a constant from the `ErrorCode` enum).
- `Error.func`: the name of the function that raised the exception.
- `Error.arguments`: a dictionary containing the arguments the function was called with.

5 FUNCTION REFERENCE

5.1 Initialization and Deinitialization

5.1.1 FindControllers

Description

Searches communication interfaces for MCS2 controllers. Returns a list of locators (see 2.3.2) for all controllers that could be detected.

C/C++

```
SA_MC_Result SA_MC_FindControllers(const char *options,
                                   char *outBuffer,
                                   size_t *bufferSize);
```

Python

```
locators = smaract.mc.FindControllers(options = "",bufferSize = 256)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

options
A string with options (currently unused). In Python, the options string is set to a default value.

outBuffer (only C/C++)
The string containing the locators of the MCS2 controllers if the function returns without error. Multiple locators are separated with a newline character.

bufferSize
The size of the buffer provided by the caller.

In Python, the buffer is allocated internally. The buffer size is an input parameter only and is set to a default value.

5.1.2 Open

Description

`Open` initializes a device. If the initialization was successful, a *device handle* is returned. The handle must be passed to other API functions. It remains valid until `Close` is called for that handle. See also 2.4.

C/C++

```
SA_MC_Result SA_MC_Open(SA_MC_Handle *outHandle, const char *options)
```

Python

```
outHandle = smaract.mc.Open(options)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

outHandle
The created device handle.

options
The *options* parameter is a string containing options for device initialization. Multiple options must be separated by a newline character:

`"model 31001\nlocator usb:sn:MCS2-00001234"`

Possible options:

- `locator <locator>`: the locator that identifies the MCS2 controller, see 2.3. (required option)
- `model <code>`: the model code of the device. (required option)

5.1.3 Close

Description

Closes a device. Invalidates the handle.

C/C++

```
SA_MC_Result SA_MC_Close(SA_MC_Handle handle)
```

Python

```
smaract.mc.Close(handle)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

5.2 Event Handling

5.2.1 WaitForEvent

Description

Blocks the caller until an *event notification* is available. For details about handling event notifications and usage examples see section 2.8.

C/C++

```
SA_MC_Result SA_MC_WaitForEvent(SA_MC_Handle handle,
                                SA_MC_Event *outEvent, int32_t timeout)
```

Python

```
outEvent = smaract.mc.WaitForEvent(handle, timeout)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

outEvent
A data object containing the event type and, depending on the type, further information about the event, see 4.1.4.

timeout
The maximum duration in milliseconds the function will block if no event notification is available. In case of a timeout the function terminates with *ERROR_TIMEOUT*. The timeout value must be positive or the constant *INFINITE*. If it is *INFINITE*, the function will never return with *ERROR_TIMEOUT*.

5.2.2 Cancel

Description

Unblocks a blocking `WaitForEvent` function call. `WaitForEvent` will return with *ERROR_CANCELED*.

C/C++

```
SA_MC_Result SA_MC_Cancel(SA_MC_Handle handle)
```

Python

```
smaract.mc.Cancel(handle)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

5.3 Motion Commands

5.3.1 Reference

Description

References all axes. The referencing procedure depends on the device class and model. Some models allow to configure the referencing behavior through properties.

The function returns immediately to the caller. If the returned result value is *ERROR_NONE*, the movement has started and the application can wait for its termination with `WaitForEvent`.

> *Please ensure that the device has sufficient travel space to avoid collisions when referencing or the device might get damaged or damage other objects!*

See also 2.5.

C/C++

```
SA_MC_Result SA_MC_Reference(SA_MC_Handle handle)
```

Python

```
smaract.mc.Reference(handle)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

5.3.2 Move

Description

Moves the device to the specified end effector target pose. The details of preparing and executing a movement are outlined in section 3.

The function returns immediately to the caller. If the returned result value is *ERROR_NONE*, the movement has started and the application can wait for its termination with `WaitForEvent`.

C/C++

```
SA_MC_Result SA_MC_Move(SA_MC_Handle handle, const SA_MC_Pose *pose)
```

Python

```
smaract.mc.Move(handle, pose)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

pose
The target pose, see 4.1.5.

5.3.3 Stop

Description

Stops a movement initiated by `Move` or `Reference` or stops the active holding of the target position of the last movement, if that movement was started with a *HOLD_TIME* property value > 0.

C/C++

```
SA_MC_Result SA_MC_Stop(SA_MC_Handle handle)
```

Python

```
smaract.mc.Stop(handle)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

5.4 Poses

5.4.1 GetPose

Description

Returns the current pose of the device.
Returns with error *NOT_REFERENCED* if the device is not referenced.

C/C++

```
SA_MC_Result SA_MC_GetPose(SA_MC_Handle handle, SA_MC_Pose *pose)
```

Python

```
pose = smaract.mc.GetPose(handle)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

pose
The returned pose.

5.4.2 SetPivot

Description

Sets the pivot point (the center of rotations).

C/C++

```
SA_MC_Result SA_MC_SetPivot(SA_MC_Handle handle, const SA_MC_Vec3 *pivot)
```

Python

```
smaract.mc.SetPivot(handle,pivot)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

pivot
A 3-tuple of (*x*,*y*,*z*). *x*, *y* and *z* are coordinates in meters.

5.4.3 GetPivot

Description

Returns the current pivot point. See `SetPivot`.

C/C++

```
SA_MC_Result SA_MC_GetPivot(SA_MC_Handle handle, SA_MC_Vec3 *pivot)
```

Python

```
smaract.mc.SetPivot(handle, pivot)
```

The Python function may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The device handle.

pivot
A 3-tuple of (*x*,*y*,*z*). *x*, *y* and *z* are coordinates in meters.

5.5 Properties

5.5.1 Property Get and Set

Description

Several function exist for reading and writing property values of different data types. See 2.7 for more information about the concept of properties.

C/C++

```
SA_MC_Result SA_MC_GetProperty_i32(SA_MC_Handle handle, SA_MC_PropertyKey pkey,
                                   int32_t *i32Value);
SA_MC_Result SA_MC_SetProperty_i32(SA_MC_Handle handle, SA_MC_PropertyKey pkey,
                                   int32_t i32Value);

SA_MC_Result SA_MC_GetProperty_f64(SA_MC_Handle handle, SA_MC_PropertyKey pkey,
                                   double *doubleValue);
SA_MC_Result SA_MC_SetProperty_f64(SA_MC_Handle handle, SA_MC_PropertyKey pkey,
                                   double doubleValue);

SA_MC_Result SA_MC_GetProperty_s(SA_MC_Handle handle, SA_MC_PropertyKey pkey,
                                 char *outBuffer, size_t *bufferSize);
```

Python

```
i32Value = smaract.mc.GetProperty_i32(handle, pkey)
smaract.mc.SetProperty_i32(handle, pkey, i32value)

doubleValue = smaract.mc.GetProperty_f64(handle, pkey)
smaract.mc.SetProperty_f64(handle, pkey, doubleValue)

stringValue = smaract.mc.GetProperty_s(handle, pkey, bufferSize=256)
```

The Python functions may raise an exception of type `smaract.mc.Error` (see 4.2.1).

Parameters

handle
The handle.

i32Value, doubleValue, stringValue
Values of the data type that matches the respective Get/Set function.

outBuffer (only C/C++)
The string containing the locators of the MCS2 controllers if the function returns without error. Multiple locators are separated with a newline character.

bufferSize
The size of the buffer provided by the caller. In Python, the buffer is allocated internally. The buffer size is an input parameter only and is set to a default value.

5.6 Common Functions

5.6.1 GetResultInfo

Description

Returns a string representation of the result code.

C/C++

```
SA_MC_Result SA_MC_GetResultInfo(SA_MC_Result resultCode, const char **info);
```

Python

```
description = smaract.mc.GetResultInfo(resultCode)
```

Parameters

resultCode
A result code.

6 EVENT REFERENCE

6.1.1 MOVEMENT_FINISHED

Event Causes

- All axes involved in a movement operation have reached their target positions.
- All or some axes failed to reach their target.
- A movement was interrupted by a `Stop` command.

Event Data Fields

- **type**: *MOVEMENT_FINISHED*
- **i32**: reason of the movement termination (result code).

7 PROPERTY REFERENCE

This chapter lists all common properties. Device class specific properties are not included here.

Access: "r/w" indicates that this property is readable and writable. "r/o" indicates read-only access.

7.1 Device Properties Reference

7.1.1 MODEL_CODE

Key	Data Type	Access	Applicable	Values
`0x0a02`	`i32`	`r/o`	`Positioners`	`positive integers`

The model code of the positioning device.

7.1.2 MODEL_NAME

Key	Data Type	Access	Applicable	Values
`0x0a03`	`string`	`r/o`	`Positioners`	`single line string`

The name of the model.

7.1.3 IS_REFERENCED

Key	Data Type	Access	Applicable	Values
`0x2a01`	`i32`	`r/o`	`Positioners`	`TRUE, FALSE`

The property is *TRUE* if all device axes have been referenced and the controller knows their current absolute positions. Otherwise the property value is *FALSE*.

7.1.4 HOLD_TIME

Key	Data Type	Access	Applicable	Values
`0x2000`	`i32`	`r/w`	`Positioners`	`0, positive integers or INFINITE`

The time the axes actively hold their target positions after a successful completion of a movement, in milliseconds. Set value to *INFINITE* to hold the positions indefinitely.

7.1.5 MAX_SPEED_LINEAR_AXES

Key	Data Type	Access	Applicable	Values
`0x2010`	`f64`	`r/w`	`Positioners`	`> 0`

Maximum speed for all linear axes in m/s.

7.1.6 MAX_SPEED_ROTARY_AXES

Key	Data Type	Access	Applicable	Values
`0x2011`	`f64`	`r/w`	`Positioners`	`> 0`

Maximum speed for all rotary axes in °/s.

7.1.7 PIEZO_MAX_CLF_LINEAR_AXES

Key	Data Type	Access	Applicable	Values
`0x2020`	`i32`	`r/w`	`Positioners`	`[0,...,20000]`

Maximum closed-loop frequency of linear piezo axes in Hz.

7.1.8 PIEZO_MAX_CLF_ROTARY_AXES

Key	Data Type	Access	Applicable	Values
`0x2021`	`i32`	`r/w`	`Positioners`	`[0,...,20000]`

Maximum closed-loop frequency of rotary piezo axes in Hz.

8 ERROR CODE REFERENCE

ERROR_NONE (0x0000)

No error. Function execution was successful.

ERROR_OTHER (0x0001)

Unspecific error. This could indicate a software bug.

ERROR_INVALID_PARAMETER (0x0002)

Invalid parameter in function call.

ERROR_INVALID_LOCATOR (0x0003)

Invalid locator in call to `Open` function.

ERROR_INVALID_HANDLE (0x0005)

Invalid handle used in function call.

ERROR_NOT_SUPPORTED (0x0006)

Tried to use an unspported feature.

ERROR_SOFTWARE_RESOURCE_LIMIT (0x0007)

A software resource limit was reached.

ERROR_QUERYBUFFER_SIZE (0x0008)

The supplied buffer is too small.

ERROR_WRONG_DATA_TYPE (0x0009)

Tried to access data with function of wrong data type.

ERROR_NO_ACCESS (0x000a)

Access not possible, e.g. when writing to a non-writable resource.

ERROR_INVALID_PROPERTY (0x0020)

The property is undefined.

ERROR_INVALID_META_PROPERTY (0x0021)

The meta property is undefined.

ERROR_CANCELED (0x0100)

An operation has been canceled while waiting for a result. Returned by `WaitForEvent` if the `Cancel` function is called while the function is waiting.

ERROR_TIMEOUT (0x0101)

An operation has timed out.

ERROR_POSE_UNREACHABLE (0x0200)

The pose specified in the Move command cannot be reached by the positioning device.

ERROR_NOT_REFERENCED (0x0201)

The device has not been referenced.

ERROR_BUSY (0x0203)

An operation could not be started because the device is busy.

ERROR_ENDSTOP_REACHED (0x0300)

Axes were blocked during movement.

ERROR_FOLLOWING_ERROR_LIMIT_REACHED (0x0301)

The following error limit has been exceeded during movement.

ERROR_REFERENCING_FAILED (0x0320)

Referencing has failed.

ERROR_DRIVER_FAILED (0x0500)

Could not load required a required driver.

ERROR_CONNECT_FAILED (0x0501)

Could not connect to an axis controller.

ERROR_NOT_CONNECTED (0x0502)

Operation could not be started because there is no connection with the axis controller.

ERROR_CONTROLLER_CONFIGURATION (0x0503)

Software could not connect to the axis controller because the controller has a configuration incompatible with the positioning device.

ERROR_COMMUNICATION_FAILED (0x0504)

Error when communicating with the axis controller.