

API Reference

1. **mod** AIRBOT Python SDK

AIRBOT Python SDK

Modules:

Name	Description
<code>arm</code>	This is the Python SDK for AIRBOT arm product series (AIRBOT Play, AIRBOT Pro, AIRBOT Play Lite

1.1 **mod** `airbot_py.arm`

This is the Python SDK for AIRBOT arm product series (AIRBOT Play, AIRBOT Pro, AIRBOT Play Lite and AIRBOT Replay). SDK for AIRBOT series is a thin gRPC client to communicate with driver.

The examples can also run with shortcuts:

Example	Shortcut	Launch via Python module
Keyboard control	<code>arm_kbd_ctrl</code>	<code>python3 -m airbot_examples.task_kbd_ctrl</code>
Motion following	<code>arm_follow</code>	<code>python3 -m airbot_examples.task_follow</code>
Switch control mode	<code>arm_switch_mode</code>	<code>python3 -m airbot_examples.switch_mode</code>
Load app	<code>arm_load_app</code>	<code>python3 -m airbot_examples.app_load</code>
Unload app	<code>arm_unload_app</code>	<code>python3 -m airbot_examples.app_unload</code>
Print joint position	<code>arm_joint_state</code>	<code>python3 -m airbot_examples.get_joint_pos</code>
Print end pose	<code>arm_end_pose</code>	<code>python3 -m airbot_examples.get_end_pose</code>
Parameter query	<code>arm_get_params</code>	<code>python3 -m airbot_examples.get_end_pose</code>
Parameter update	<code>arm_set_params</code>	<code>python3 -m airbot_examples.set_params</code>

Example	Shortcut	Launch via Python module
Move end effector to a cartesian pose	<code>arm_move_cart_pose</code>	<code>python3 -m airbot_examples.move_cart_pose</code>
Move to a set of joint angles	<code>arm_move_joint</code>	<code>python3 -m airbot_examples.move_joint_pos</code>
Move along a set of cartesian waypoints	<code>arm_move_cart_waypoints</code>	<code>python3 -m airbot_examples.move_with_cart_waypoints</code>
Move along a set of joint waypoints	<code>arm_move_joint_waypoints</code>	<code>python3 -m airbot_examples.move_with_joint_waypoints</code>
Swing example: continuous swing in joint space	<code>arm_example_swing</code>	<code>python3 -m airbot_examples.task_swing</code>
Wipe example: continuous wipe in cartesian space	<code>arm_example_wipe</code>	<code>python3 -m airbot_examples.task_wipe</code>

Classes:

Name	Description
<code>AIRBOTArm</code>	The client instance for AIRBOT arm series.
<code>RobotMode</code>	Enum for the control mode of the arm.
<code>SpeedProfile</code>	Enum for speed profile of the robot.
<code>State</code>	Enum for the state of the driver service.

Attributes:

Name	Type	Description
<code>AIRBOTPlay</code>		Alias for <code>AIRBOTArm</code>
<code>AIRBOTPlayLite</code>		Alias for <code>AIRBOTArm</code>

Name	Type	Description
<code>AIRBOTPro</code>		Alias for <code>AIBROTArm</code>

1.1.1 `attr` `airbot_py.arm.AIRBOTPlay = AIRBOTArm` module-attribute

Alias for `AIBROTArm`

1.1.2 `attr` `airbot_py.arm.AIRBOTPlayLite = AIRBOTArm` module-attribute

Alias for `AIBROTArm`

1.1.3 `attr` `airbot_py.arm.AIRBOTPro = AIRBOTArm` module-attribute

Alias for `AIBROTArm`

1.1.4 `class` `airbot_py.arm.AIRBOTArm(url='localhost', port=50051)`

The client instance for AIRBOT arm series.

Initialize SDK instance with default values. Would NOT perform any actions including connecting to server.

Parameters:

Name	Type	Description	Default
<code>url</code>	<code>str</code>	The listening url for the driver server. Default: "localhost"	<code>'localhost'</code>
<code>port</code>	<code>int</code>	The listening port for the driver server. Default: 50051	<code>50051</code>

Methods:

Name	Description
<code>connect</code>	Connect to the server if currently not connected.
<code>disconnect</code>	Disconnect from to the server if currently not connected.
<code>get_control_mode</code>	Obtain current control mode of the driver service.
<code>get_eef_eff</code>	Read current end effector torque output (in Nm) from the SDK server.
<code>get_eef_pos</code>	Read current end effector width (in meters) from the SDK server.
<code>get_end_pose</code>	Read current end pose from the SDK server.

Name	Description
<code>get_joint_eff</code>	Read current joint efforts (in Nm) from the SDK server.
<code>get_joint_pos</code>	Read current joint positions (in radian) from the SDK server.
<code>get_joint_vel</code>	Read current joint velocities (in radian/s) from the SDK server.
<code>get_params</code>	Get parameters from the driver server.
<code>get_state</code>	Obtain current state of the driver service.
<code>load_app</code>	Load an app to the driver server.
<code>mit_joint_integrated_control</code>	Send integrated 6 joint position command to the robot.
<code>move_eef_pos</code>	Send one joint position command to the end effector.
<code>move_eff_pos</code>	Previously name with typo for sending the command to the end effector.
<code>move_to_cart_pose</code>	Send one set of cartesian pose command to the robot.
<code>move_to_joint_pos</code>	Send one set of joint position command to the robot.
<code>move_with_cart_waypoints</code>	Move the robot arm along a series of cartesian waypoints.
<code>move_with_joint_waypoints</code>	Move the robot arm along a series of joint space waypoints.
<code>servo_cart_pose</code>	Send continuous cartesian pose commands to the robot.
<code>servo_cart_twist</code>	Send continuous cartesian twist commands to the robot.
<code>servo_eef_force</code>	UNIMPLEMENTED — This method has been removed and will be replaced.
<code>servo_eef_pos</code>	Send continuous end effector width commands to the robot.
<code>servo_joint_pos</code>	Send continuous joint position commands to the robot.
<code>servo_joint_vel</code>	Send continuous joint velocity commands to the robot.
<code>set_params</code>	Set parameters to the driver server.
<code>set_speed_profile</code>	Set the speed profile of the robot.

Name	Description
<code>switch_mode</code>	Switch the control mode of the robot.
<code>unload_app</code>	Unload the app from the driver server.

Source code in `airbot_py/arm.py`

```

156 def __init__(self, url: str = "localhost", port: int = 50051):
157     """
158     Initialize SDK instance with default values.
159     Would NOT perform any actions including connecting to server.
160
161     Args:
162         url (str): The listening url for the driver server. Default: "localhost"
163         port (int): The listening port for the driver server. Default: 50051
164     """
165     self.endpoint = url + ":" + str(port)
166     self.logger = logging.getLogger(__name__)
167     self.client_id = str(uuid.uuid4())
168     self.gap = 1 / 250
169     self.logger.info("Client ID: %s", self.client_id)
170     self._control_stub = None
171     self._feedback_stub = None
172     self._utils_stub = None
173     self._channel = None
174     self._feedback_jointstates = None
175     self._feedback_end_poses = None
176     self._feedback_thread = None
177     self._servo_controller = None
178     self._mit_controller = None
179     self._end_joint_states = None
180     self._feedbacking = False

```

1.1.4.1 METH `AIRBOT_PY.ARM.AIRBOTARM.CONNECT()`

Connect to the server if currently not connected.

Directly calling this method is deprecated in favor of using the `with` statement:

```

# Use:
with AIRBOTArm() as robot:
    ...

# instead of
robot = AIRBOTArm()
robot.connect()
...
robot.disconnect()

```

This method is idempotent. Calling this method if SDK has already connected to the server would cause no harm.

Returns:

Name	Type	Description
result	bool	True if connected successfully

```

422 def connect(self) -> bool:
423     """
424     Connect to the server if currently not connected.
425
426     Directly calling this method is deprecated in favor of using the `with` statement:
427
428     ```python
429     # Use:
430     with AIRBOTArm() as robot:
431         ...
432
433     # instead of
434     robot = AIRBOTArm()
435     robot.connect()
436     ...
437     robot.disconnect()
438     ```
439
440     This method is idempotent. Calling this method if SDK has already
441     connected to the server would cause no harm.
442
443     Returns:
444         result (bool): True if connected successfully
445     """
446     if self._control_stub is None:
447         self._channel = grpc.insecure_channel(self.endpoint)
448         self._control_stub = arm_pb2_grpc.ControlServiceStub(self._channel)
449         self._feedback_stub = arm_pb2_grpc.FeedbackServiceStub(self._channel)
450         self._utils_stub = utils_pb2_grpc.UtilsServiceStub(self._channel)
451
452         self._feedbacking = True
453         self._feedback_thread = threading.Thread(target=self._status)
454         self._feedback_thread.start()
455         start_time = time.time()
456         while self._feedback_jointstates is None:
457             time.sleep(0.01)
458             if time.time() - start_time > 1:
459                 logging.error("Failed to connect to the server.")
460                 self._feedbacking = False
461                 self._feedback_thread.join()
462                 self._feedback_thread = None
463                 self._control_stub = None
464                 self._feedback_stub = None
465                 self._utils_stub = None
466                 if self._channel is not None:
467                     self._channel.close()
468                 self._channel = None
469                 return False
470
471         self._servo_controller = ServoController(
472             self.client_id, self._control_stub, self.logger
473         )
474
475         return True
476     else:
477         logging.warning("Already connected to the server.")
478         return True

```

Disconnect from to the server if currently not connected.

Directly calling this method is deprecated in favor of using the `with` statement:

```
# Use:
with AIRBOTArm() as robot:
    ...

# instead of
robot = AIRBOTArm()
robot.connect()
...
robot.disconnect()
```

This method is idempotent. Calling this method if SDK has already disconnected from the server would cause no harm.

Returns:

Name	Type	Description
result	bool	True if connected successfully

```

480 def disconnect(self) -> bool:
481     """
482     Disconnect from to the server if currently not connected.
483
484     Directly calling this method is deprecated in favor of using the `with` statement:
485
486     ```python
487     # Use:
488     with AIRBOTArm() as robot:
489         ...
490
491     # instead of
492     robot = AIRBOTArm()
493     robot.connect()
494     ...
495     robot.disconnect()
496     ```
497
498     This method is idempotent. Calling this method if SDK has already
499     disconnected from the server would cause no harm.
500
501     Returns:
502         result(bool): True if connected successfully
503     """
504     if self._feedbacking:
505         self._feedbacking = False
506         if self._feedback_thread is not None:
507             self._feedback_thread.join()
508             self._feedback_thread = None
509
510     if self._servo_controller is not None:
511         self._servo_controller.stop()
512         self._servo_controller = None
513
514     if self._mit_controller is not None:
515         self._mit_controller.stop()
516         self._mit_controller = None
517
518     if self._control_stub is not None:
519         self._control_stub = None
520         self._feedback_stub = None
521         self._utils_stub = None
522         self._channel.close()
523         self._channel = None
524         return True
525     else:
526         logging.warning("Already disconnected from the server.")
527         return True

```

1.1.4.3 METH `AIRBOT_PY.ARM.AIRBOTARM.GET_CONTROL_MODE()`

Obtain current control mode of the driver service.

Returns:

Name	Type	Description
------	------	-------------

Name	Type	Description
mode	RobotMode	The current control mode of the driver service.

Source code in `airbot_py/arm.py`

```

1118 @grpc_safe()
1119 @typechecked
1120 def get_control_mode(self) -> Optional[RobotMode]:
1121     """
1122     Obtain current control mode of the driver service.
1123
1124     Returns:
1125         mode (RobotMode): The current control mode of the driver service.
1126     """
1127     stamp = time.time_ns()
1128     request = arm_pb2.GetModeRequest()
1129     request.stamp.sec = int(stamp / 1e9)
1130     request.stamp.nanosec = int(stamp % 1e9)
1131     request.request_id = self.client_id
1132     request.component_names[:] = ["arm"]
1133
1134     response = self._feedback_stub.GetMode(request)
1135     return RobotMode(response.modes[0]) if response.modes else None

```

1.1.4.4 `METH AIRBOT_PY.ARM.AIRBOTARM.GET_EEF_EFF()`

Read current end effector torque output (in Nm) from the SDK server. Currently only parallel two-finger end effector is supported.

If no end effector is installed, an empty list is returned.

If the SDK has not received joint states from the server, None is returned.

Returns:

Name	Type	Description
eef_eff	list[float]	the current end effector torque output of the arm. None if SDK has not received joint states or no end effector is installed.

```

345 @grpc_safe()
346 def get_eef_eff(self) -> Optional[List[float]]:
347     """
348     Read current end effector torque output (in Nm) from the SDK server.
349     Currently only parallel two-finger end effector is supported.
350
351     If no end effector is installed, an empty list is returned.
352
353     If the SDK has not received joint states from the server, None is returned.
354
355     Returns:
356         eef_eff (list[float]): the current end effector torque output of the arm. \
357         None if SDK has not received joint states or no end \
358         effector is installed.
359     """
360     if self._feedback_jointstates is None:
361         logging.error("No joint states received yet.")
362         return None
363     else:
364         return [
365             i[1]
366             for i in sorted(
367                 (
368                     (name, effort)
369                     for name, effort in zip(
370                         self._feedback_jointstates.name,
371                         self._feedback_jointstates.effort,
372                     )
373                 ),
374                 key=lambda x: x[0],
375             )
376             if i[0] == "gripper_mapping_controller"
377         ]

```

1.1.4.5 METH AIRBOT_PY.ARM.AIRBOTARM.GET_EEF_POS()

Read current end effector width (in meters) from the SDK server. Currently only parallel two-finger end effector is supported.

If no end effector is installed, an empty list is returned.

If the SDK has not received joint states from the server, None is returned.

Returns:

Name	Type	Description
<code>eef_pos</code>	<code>list[float]</code> <code>None</code>	the current end effector width of the arm. None if SDK has not received joint states or no end effector is installed.

Source code in `airbot_py/arm.py`

```
247 @grpc_safe()
248 def get_eef_pos(self) -> Optional[List[float]]:
249     """
250     Read current end effector width (in meters) from the SDK server.
251     Currently only parallel two-finger end effector is supported.
252
253     If no end effector is installed, an empty list is returned.
254
255     If the SDK has not received joint states from the server, None is returned.
256
257     Returns:
258         eef_pos (list[float] | None): the current end effector width of the arm. \
259         None if SDK has not received joint states or no end \
260         effector is installed.
261     """
262     if self._feedback_jointstates is None:
263         logging.error("No joint states received yet.")
264         return None
265     else:
266         return [
267             i[1]
268             for i in sorted(
269                 (
270                     (name, position)
271                     for name, position in zip(
272                         self._feedback_jointstates.name,
273                         self._feedback_jointstates.position,
274                     )
275                 ),
276                 key=lambda x: x[0],
277             )
278             if i[0] == "gripper_mapping_controller"
279         ]
```

1.1.4.6 METH AIRBOT_PY.ARM.AIRBOTARM.GET_END_POSE()

Read current end pose from the SDK server.

The zero point of the reference frame for the end pose is the intersection point of the rotating axis of first joint and installing plane. The axes of reference frame for the end are x axis pointing forward, y axis pointing leftward and z axis pointing upward.

If no end effector is installed, the pose of the end flange is returned. If any end effector is installed, the pose of the effecting point is returned. For example, the effecting point of AIRBOT G2 is the grasping point in the front.

TODO: Replace with actual doc url For detailed explanation of the reference frame and the end link used for get_end_pose, please refer to <https://www.bing.com>

Returns:

Name	Type	Description
<code>end_pose</code>	<code>list[list[float]]</code>	A list of two float-list. The first list is the cartesian translation of the pose in

Name

Type

Description

meters, and the second list is the orientation quaternion (x, y, z, w) of the pose.

Source code in `airbot_py/arm.py`

```
379 @grpc_safe()
380 def get_end_pose(self) -> Optional[List[List[float]]]:
381     """
382     Read current end pose from the SDK server.
383
384     The zero point of the reference frame for the end pose is the intersection point
385     of the rotating axis of first joint and installing plane.
386     The axes of reference frame for the end are x axis pointing forward, y axis
387     pointing leftward and z axis pointing upward.
388
389     If no end effector is installed, the pose of the end flange is returned.
390     If any end effector is installed, the pose of the effecting point
391     is returned. For example, the effecting point of AIRBOT G2 is the grasping
392     point in the front.
393
394     TODO: Replace with actual doc url
395     For detailed explanation of the reference frame and the end link used
396     for get_end_pose, please refer to [https://www.bing.com](https://www.bing.com)
397
398     Returns:
399         end_pose (list[list[float]]): A list of two float-list. The first list is \
400         the cartesian translation of the pose in meters, and the second list \
401         is the orientation quaternion (x, y, z, w) of the pose.
402
403     """
404     if self._feedback_end_poses is None:
405         logging.error("No end poses received yet.")
406         return None
407     else:
408         return [
409             [
410                 self._feedback_end_poses.translation.x,
411                 self._feedback_end_poses.translation.y,
412                 self._feedback_end_poses.translation.z,
413             ],
414             [
415                 self._feedback_end_poses.orientation.x,
416                 self._feedback_end_poses.orientation.y,
417                 self._feedback_end_poses.orientation.z,
418                 self._feedback_end_poses.orientation.w,
419             ],
420         ]
```

1.1.4.7 METH AIRBOT_PY.ARM.AIRBOTARM.GET_JOINT_EFF()

Read current joint efforts (in Nm) from the SDK server.

The returned list only contains the joint efforts of the arm itself. The position of the end effector (if any) is not included.

If SDK has not received joint states from the server, None is returned.

Returns:

Name	Type	Description
joint_eff	float None	the current efforts of the arm joints.

Source code in airbot_py/arm.py

```
313 @grpc_safe()
314 def get_joint_eff(self) -> Optional[List[float]]:
315     """
316     Read current joint efforts (in Nm) from the SDK server.
317
318     The returned list only contains the joint efforts of the arm itself.
319     The position of the end effector (if any) is not included.
320
321     If SDK has not received joint states from the server, None is returned.
322
323     Returns:
324         joint_eff (float | None): the current efforts of the arm joints.
325     """
326     if self._feedback_jointstates is None:
327         logging.error("No joint states received yet.")
328         return None
329     else:
330         return [
331             i[1]
332             for i in sorted(
333                 (
334                     (name, effort)
335                     for name, effort in zip(
336                         self._feedback_jointstates.name,
337                         self._feedback_jointstates.effort,
338                     )
339                 ),
340                 key=lambda x: x[0],
341             )
342             if i[0].startswith("joint")
343         ]
```

1.1.4.8 METH AIRBOT_PY.ARM.AIRBOTARM.GET_JOINT_POS()

Read current joint positions (in radian) from the SDK server.

The returned list only contains the joint positions of the arm itself. The position of the end effector (if any) is not included.

If SDK has not received joint states from the server, None is returned

Returns:

Name	Type	Description
joint_pos	list[float] None	the current positions of the arm joints.

Source code in `airbot_py/arm.py`

```
215 @grpc_safe()
216 def get_joint_pos(self) -> Optional[List[float]]:
217     """
218     Read current joint positions (in radian) from the SDK server.
219
220     The returned list only contains the joint positions of the arm itself.
221     The position of the end effector (if any) is not included.
222
223     If SDK has not received joint states from the server, None is returned
224
225     Returns:
226         joint_pos (list[float] | None): the current positions of the arm joints.
227     """
228     if self._feedback_jointstates is None:
229         logging.error("No joint states received yet.")
230         return None
231     else:
232         return [
233             i[1]
234             for i in sorted(
235                 (
236                     (name, position)
237                     for name, position in zip(
238                         self._feedback_jointstates.name,
239                         self._feedback_jointstates.position,
240                     )
241                 ),
242                 key=lambda x: x[0],
243             )
244             if i[0].startswith("joint")
245         ]
```

1.1.4.9 METH AIRBOT_PY.ARM.AIRBOTARM.GET_JOINT_VEL()

Read current joint velocities (in radian/s) from the SDK server.

The returned list only contains the joint velocities of the arm itself. The position of the end effector (if any) is not included.

If SDK has not received joint states from the server, None is returned

Returns:

Name	Type	Description
<code>joint_vel</code>	<code>list[float] None</code>	the current velocities of the arm joints.

Source code in `airbot_py/arm.py`

```
281 @grpc_safe()
282 def get_joint_vel(self) -> Optional[List[float]]:
283     """
284     Read current joint velocities (in radian/s) from the SDK server.
285
286     The returned list only contains the joint velocities of the arm itself.
287     The position of the end effector (if any) is not included.
288
289     If SDK has not received joint states from the server, None is returned
290
291     Returns:
292         joint_vel (list[float] | None): the current velocities of the arm joints.
293     """
294     if self._feedback_jointstates is None:
295         logging.error("No joint states received yet.")
296         return None
297     else:
298         return [
299             i[1]
300             for i in sorted(
301                 (
302                     (name, velocity)
303                     for name, velocity in zip(
304                         self._feedback_jointstates.name,
305                         self._feedback_jointstates.velocity,
306                     )
307                 ),
308                 key=lambda x: x[0],
309             )
310             if i[0].startswith("joint")
311         ]
```

1.1.4.10 METH `AIRBOT_PY.ARM.AIRBOTARM.GET_PARAMS(NAMES)`

Get parameters from the driver server.

Parameters:

Name	Type	Description	Default
<code>names</code>	<code>list[str]</code>	The names of the parameters to get.	<i>required</i>

Returns: `params (dict)`: A dictionary of the parameters.

```

1171 @grpc_safe()
1172 @typechecked
1173 def get_params(self, names: List[str]) -> Dict[str, Any]:
1174     """
1175     Get parameters from the driver server.
1176
1177     Args:
1178         names (list[str]): The names of the parameters to get.
1179     Returns:
1180         params (dict): A dictionary of the parameters.
1181     """
1182     if len(names) == 0:
1183         logging.error("No parameter names provided.")
1184         return {}
1185     stamp = time.time_ns()
1186     request = arm_pb2.GetParamsRequest()
1187     request.stamp.sec = int(stamp / 1e9)
1188     request.stamp.nanosec = int(stamp % 1e9)
1189     request.request_id = self.client_id
1190     request.keys[:] = names
1191
1192     response = self._feedback_stub.GetParams(request)
1193     return {
1194         k: getattr(v, v.WhichOneof("union"))
1195         for k, v in zip(response.keys, response.values)
1196     }

```

1.1.4.11 METH AIRBOT_PY.ARM.AIRBOTARM.GET_STATE()

Obtain current state of the driver service.

Returns:

Name	Type	Description
<code>state</code>	<code>State</code>	The current state of the driver service.

Source code in `airbot_py/arm.py`

```
1100 @grpc_safe()
1101 @typechecked
1102 def get_state(self) -> State:
1103     """
1104     Obtain current state of the driver service.
1105
1106     Returns:
1107         state (State): The current state of the driver service.
1108     """
1109     stamp = time.time_ns()
1110     request = utils_pb2.GetStateRequest()
1111     request.stamp.sec = int(stamp / 1e9)
1112     request.stamp.nanosec = int(stamp % 1e9)
1113     request.request_id = self.client_id
1114
1115     response = self._utils_stub.GetState(request)
1116     return State(response.state)
```

1.1.4.12 **METH** `AIRBOT_PY.ARM.AIRBOTARM.LOAD_APP(NAME, PARAMS=NONE)`

Load an app to the driver server.

When an read-write app is loaded, the SDK would enter read-only state, in which get methods are available but set methods are blocked. See [available states](#)

Currently available apps: * `record_replay_app/record_replay_app::RecordReplayApp`

Parameters:

Name	Type	Description	Default
<code>name</code>	<code>str</code>	The name of the app to load.	<i>required</i>
<code>params</code>	<code>dict</code>	The parameters for the app. Default: {}	<code>None</code>

Returns:

Name	Type	Description
<code>result</code>	<code>bool</code>	True if the app was loaded successfully, False otherwise.

Source code in `airbot_py/arm.py`

```
1137 @grpc_safe()
1138 @typechecked
1139 def load_app(self, name: str, params: Optional[dict] = None) -> bool:
1140     """
1141     Load an app to the driver server.
1142
1143     When an read-write app is loaded, the SDK would enter read-only state,
1144     in which get methods are available but set methods are blocked. See
1145     [available states](./#airbot_py.arm--internal-states-of-the-driver)
1146
1147     Currently available apps:
1148     * `record_replay_app/record_replay_app::RecordReplayApp`
1149
1150     Args:
1151         name (str): The name of the app to load.
1152         params (dict): The parameters for the app. Default: {}
1153
1154     Returns:
1155         result (bool): True if the app was loaded successfully, False otherwise.
1156     """
1157     stamp = time.time_ns()
1158     request = utils_pb2.LoadAppRequest()
1159     request.stamp.sec = int(stamp / 1e9)
1160     request.stamp.nanosec = int(stamp % 1e9)
1161     request.request_id = self.client_id
1162     request.app_name = name
1163     if params is not None:
1164         for key, value in params.items():
1165             request.params_key.append(key)
1166             request.params_value.append(str(value))
1167
1168     response = self._utils_stub.LoadApp(request)
1169     return response.ok
```

1.1.4.13 **METH** `AIRBOT_PY.ARM.AIRBOTARM.MIT_JOINT_INTEGRATED_CONTROL(JOINT_POS, JOINT_VEL, JOINT_EFF, JOINT_KP, JOINT_KD)`

Send integrated 6 joint position command to the robot. This method should run in `MIT_INTEGRATED` mode.

The joint position command is sent to the driver server, and the robot would in `MIT_INTEGRATED` mode.

```

1059 @grpc_safe()
1060 @typechecked
1061 def mit_joint_integrated_control(
1062     self,
1063     joint_pos: List[float],
1064     joint_vel: List[float],
1065     joint_eff: List[float],
1066     joint_kp: List[float],
1067     joint_kd: List[float],
1068 ) -> bool:
1069     """
1070     Send integrated 6 joint position command to the robot.
1071     This method should run in 'MIT_INTEGRATED' mode.
1072
1073     The joint position command is sent to the driver server, and the robot
1074     would in MIT_INTEGRATED mode.
1075     """
1076     if not all(
1077         len(param_list) == 6
1078         for param_list in [joint_pos, joint_vel, joint_eff, joint_kp, joint_kd]
1079     ):
1080         logging.error(
1081             "All joint command lists (pos, vel, eff, kp, kd) must be lists of 6 floats."
1082         )
1083         return False
1084
1085     if self._mit_controller is None:
1086         self._mit_controller = MITController(
1087             self.gap, self.client_id, self._control_stub, self.logger
1088         )
1089     self._mit_controller.start_thread(self._mit_controller.mit_cmd_gen)
1090     mit_cmd = types_pb2.MITControl()
1091     mit_cmd.name[:] = ["joint1", "joint2", "joint3", "joint4", "joint5", "joint6"]
1092     mit_cmd.position[:] = joint_pos
1093     mit_cmd.velocity[:] = joint_vel
1094     mit_cmd.trajectory[:] = joint_eff
1095     mit_cmd.kp[:] = joint_kp
1096     mit_cmd.kd[:] = joint_kd
1097     self._mit_controller.append_cmd(mit_cmd)
1098     return True

```

1.1.4.14 METH AIRBOT_PY.ARM.AIRBOTARM.MOVE_EEF_POS(JOINT_POS, BLOCKING=TRUE)

Send one joint position command to the end effector. This method should run in `PLANNING_POS` mode.

Effective for parallel two-finger end effector only.

Before calling this method, the robot should be in `PLANNING_POS` mode, otherwise the command would be ignored.

Parameters:

Name	Type	Description	Default
<code>joint_pos</code>	<code>list[float]</code>	The end effector width commands in meters.	<i>required</i>

Returns:

Name	Type	Description
result	bool	True if the command was sent successfully, False otherwise.

Source code in `airbot_py/arm.py`

```
814 @grpc_safe()
815 @typechecked
816 def move_eef_pos(
817     self,
818     joint_pos: Union[List[float], float],
819     blocking: bool = True,
820 ) -> bool:
821     """
822     Send one joint position command to the end effector.
823     This method should run in `PLANNING_POS` mode.
824
825     Effective for parallel two-finger end effector only.
826
827     Before calling this method, the robot should be in `PLANNING_POS` mode,
828     otherwise the command would be ignored.
829
830     Args:
831         joint_pos (list[float]): The end effector width commands in meters.
832
833     Returns:
834         result (bool): True if the command was sent successfully, False otherwise.
835     """
836     # For backward compatibility
837     if not isinstance(joint_pos, list):
838         joint_pos = [joint_pos]
839
840     def _generate_joint_pos(
841         joint_pos: List[float],
842     ) -> Iterator[arm_pb2.SetStateRequest]:
843         request = arm_pb2.SetStateRequest()
844         joint_states = types_pb2.JointStates()
845         now = time.time_ns()
846         request.stamp.sec = int(now / 1e9)
847         request.stamp.nanosec = int(now % 1e9)
848         request.request_id = self.client_id
849         request.component_names[:] = ["eef"]
850         joint_states.name[:] = ["eef"]
851         joint_states.position[:] = joint_pos
852         request.commands.add(joint_states=joint_states)
853         yield request
854
855     result_iter = self._control_stub.SetState(_generate_joint_pos(joint_pos))
856
857     first_result = next(result_iter, None)
858     if first_result is not None and first_result.status != arm_pb2.Status.ACCEPTED:
859         self.logger.error("Failed to move to joint position")
860         return False
861
862     return True
```

1.1.4.15 **METH** AIRBOT_PY.ARM.AIRBOTARM.MOVE_EFF_POS(JOINT_POS, BLOCKING=TRUE)

Previously name with typo for sending the command to the end effector. Renamed to `move_eef_pos`, but kept for backward compatibility.

Source code in `airbot_py/arm.py`

```
801 @grpc_safe()
802 @typechecked
803 def move_eff_pos(
804     self,
805     joint_pos: Union[List[float], float],
806     blocking: bool = True,
807 ) -> bool:
808     """
809     Previously name with typo for sending the command to the end effector.
810     Renamed to `move_eef_pos`, but kept for backward compatibility.
811     """
812     return self.move_eef_pos(joint_pos, blocking)
```

1.1.4.16 **METH** AIRBOT_PY.ARM.AIRBOTARM.MOVE_TO_CART_POSE(CART_POSE, BLOCKING=TRUE)

Send one set of cartesian pose command to the robot. This method should run in `PLANNING_POS` mode.

The cartesian pose command is sent to the driver server, and the robot would plan a path to reach the target cartesian pose then execute it.

The zero point of the reference frame for the end pose is the intersection point of the rotating axis of first joint and installing plane. The axes of reference frame for the end are x axis pointing forward, y axis pointing leftward and z axis pointing upward.

Parameters:

Name	Type	Description	Default
<code>cart_pose</code>	<code>list[list[float]]</code>	The cartesian pose commands. The first list is the cartesian translation of the pose in meters, and the second list is the orientation quaternion (x, y, z, w) of the pose.	<i>required</i>
<code>blocking</code>	<code>bool</code>	Whether to block until the command is finished. Default:	<code>True</code>

Returns:

Name	Type	Description
<code>result</code>	<code>bool</code>	True if the command was sent successfully, False otherwise.

Source code in `airbot_py/arm.py`



```

864 @grpc_safe()
865 @typechecked
866 def move_to_cart_pose(
867     self,
868     cart_pose: List[List[float]],
869     blocking: bool = True,
870 ) -> bool:
871     """
872     Send one set of cartesian pose command to the robot.
873     This method should run in `PLANNING_POS` mode.
874
875     The cartesian pose command is sent to the driver server, and the robot
876     would plan a path to reach the target cartesian pose then execute it.
877
878     The zero point of the reference frame for the end pose is the intersection point
879     of the rotating axis of first joint and installing plane.
880     The axes of reference frame for the end are x axis pointing forward, y axis
881     pointing leftward and z axis pointing upward.
882
883     Args:
884         cart_pose (list[list[float]]): The cartesian pose commands. \
885             The first list is the cartesian translation of the pose in meters, \
886             and the second list is the orientation quaternion (x, y, z, w) of the pose.
887         blocking (bool): Whether to block until the command is finished. \
888             Default: True
889
890     Returns:
891         result (bool): True if the command was sent successfully, False otherwise.
892     """
893
894     def _generate_cart_pose(
895         cart_pose: List[List[float]],
896     ) -> Iterator[arm_pb2.SetStateRequest]:
897         transform = types_pb2.Transform()
898         request = arm_pb2.SetStateRequest()
899         now = time.time_ns()
900         request.stamp.sec = int(now / 1e9)
901         request.stamp.nanosec = int(now % 1e9)
902         request.request_id = self.client_id
903         request.component_names[:] = ["arm"]
904         transform.translation.x = cart_pose[0][0]
905         transform.translation.y = cart_pose[0][1]
906         transform.translation.z = cart_pose[0][2]
907         transform.orientation.x = cart_pose[1][0]
908         transform.orientation.y = cart_pose[1][1]
909         transform.orientation.z = cart_pose[1][2]
910         transform.orientation.w = cart_pose[1][3]
911         request.commands.add(transform=transform)
912         yield request
913
914     result_iter = self._control_stub.SetState(_generate_cart_pose(cart_pose))
915
916     first_result = next(result_iter)
917     if first_result.status != arm_pb2.Status.ACCEPTED:
918         self.logger.error("Failed to move to joint position")
919         return False
920
921     if blocking:
922         for result in result_iter:
923             if result.status == arm_pb2.Status.FINISHED:
924                 self.logger.info("Moving to joint position...")
925                 return True
926         return False
927     else:

```

```
927         return True
928
```

1.1.4.17 METH AIRBOT_PY.ARM.AIRBOTARM.MOVE_TO_JOINT_POS(JOINT_POS, BLOCKING=TRUE)

Send one set of joint position command to the robot. This method should run in PLANNING_POS mode.

The joint position command is sent to the driver server, and the robot would plan a path to reach the target joint position then execute it.

Before calling this method, the robot should be in PLANNING_POS mode, otherwise the command would be ignored.

Parameters:

Name	Type	Description	Default
joint_pos	list[float]	The joint position commands in radian.	required
blocking	bool	Whether to block until the command is finished. Default: True	True

Returns:

Name	Type	Description
result	bool	True if the command was sent successfully, False otherwise.

Source code in `airbot_py/arm.py`



```

737 @grpc_safe()
738 @typechecked
739 def move_to_joint_pos(
740     self,
741     joint_pos: List[float],
742     blocking: bool = True,
743 ) -> bool:
744     """
745     Send one set of joint position command to the robot.
746     This method should run in `PLANNING_POS` mode.
747
748     The joint position command is sent to the driver server, and the robot
749     would plan a path to reach the target joint position then execute it.
750
751     Before calling this method, the robot should be in `PLANNING_POS` mode,
752     otherwise the command would be ignored.
753
754     Args:
755         joint_pos (list[float]): The joint position commands in radian.
756         blocking (bool): Whether to block until the command is finished.
757             Default: True
758
759     Returns:
760         result (bool): True if the command was sent successfully, False otherwise.
761     """
762
763     def _generate_joint_pos(
764         joint_pos: List[float],
765     ) -> Iterator[arm_pb2.SetStateRequest]:
766         request = arm_pb2.SetStateRequest()
767         joint_states = types_pb2.JointStates()
768         now = time.time_ns()
769         request.stamp.sec = int(now / 1e9)
770         request.stamp.nanosec = int(now % 1e9)
771         request.request_id = self.client_id
772         request.component_names[:] = ["arm"]
773         joint_states.name[:] = [
774             "joint1",
775             "joint2",
776             "joint3",
777             "joint4",
778             "joint5",
779             "joint6",
780         ]
781         joint_states.position[:] = joint_pos
782         request.commands.add(joint_states=joint_states)
783         yield request
784
785     result_iter = self._control_stub.SetState(_generate_joint_pos(joint_pos))
786
787     first_result = next(result_iter)
788     if first_result.status != arm_pb2.Status.ACCEPTED:
789         self.logger.error("Failed to move to joint position")
790         return False
791
792     if blocking:
793         for result in result_iter:
794             if result.status == arm_pb2.Status.FINISHED:
795                 self.logger.info("Moving to joint position...")
796                 return True
797         return False
798     else:
799         return True

```

1.1.4.18 **METH** `AIRBOT_PV.ARM.AIRBOTARM.MOVE_WITH_CART_WAYPOINTS(WAYPOINTS, BLOCKING=TRUE)`

Move the robot arm along a series of cartesian waypoints. This method should run in `PLANNING_WAYPOINTS` or `PLANNING_WAYPOINTS_PATH` mode.

The robot will interpolate through the given list of poses.

Parameters:

Name	Type	Description	Default
<code>waypoints</code>	<code>list[list[list[float]]]</code>	A list of poses, where each pose is a 2-element list: [translation, orientation] - translation: list of 3 floats (x, y, z) in meters - orientation: list of 4 floats (x, y, z, w) as quaternion	<i>required</i>
<code>blocking</code>	<code>bool</code>	Whether to wait until execution is finished.	<code>True</code>

Returns:

Name	Type	Description
<code>bool</code>	<code>bool</code>	True if the command was accepted and (optionally) completed.

Source code in `airbot_py/arm.py`



```

930 @grpc_safe()
931 @typechecked
932 def move_with_cart_waypoints(
933     self,
934     waypoints: List[List[List[float]]],
935     blocking: bool = True,
936 ) -> bool:
937     """
938     Move the robot arm along a series of cartesian waypoints.
939     This method should run in `PLANNING_WAYPOINTS` or `PLANNING_WAYPOINTS_PATH` mode.
940
941     The robot will interpolate through the given list of poses.
942
943     Args:
944         waypoints (list[list[list[float]]]): A list of poses, where each pose is a
945             2-element list: [translation, orientation]
946             - translation: list of 3 floats (x, y, z) in meters
947             - orientation: list of 4 floats (x, y, z, w) as quaternion
948         blocking (bool): Whether to wait until execution is finished.
949
950     Returns:
951         bool: True if the command was accepted and (optionally) completed.
952     """
953
954     def _generate_waypoints(
955         cart_pose: List[List[List[float]]],
956     ) -> Iterator[arm_pb2.SetStateRequest]:
957         waypoints_obj = types_pb2.WayPoints()
958         request = arm_pb2.SetStateRequest()
959         now = time.time_ns()
960         request.stamp.sec = int(now / 1e9)
961         request.stamp.nanosec = int(now % 1e9)
962         request.request_id = self.client_id
963         request.component_names[:] = ["arm"]
964         for e in cart_pose:
965             transform = types_pb2.Transform()
966             transform.translation.x = e[0][0]
967             transform.translation.y = e[0][1]
968             transform.translation.z = e[0][2]
969             transform.orientation.x = e[1][0]
970             transform.orientation.y = e[1][1]
971             transform.orientation.z = e[1][2]
972             transform.orientation.w = e[1][3]
973             waypoints_obj.transform_array.extend([transform])
974
975         command = types_pb2.Command()
976         command.waypoints.CopyFrom(waypoints_obj)
977         request.commands.append(command)
978         yield request
979
980     result_iter = self._control_stub.SetState(_generate_waypoints(waypoints))
981     first_result = next(result_iter)
982     if first_result.status != arm_pb2.Status.ACCEPTED:
983         self.logger.error("Failed to move to joint position")
984         return False
985     if blocking:
986         print("Waiting for arm to finish moving...")
987         for result in result_iter:
988             if result.status == arm_pb2.Status.FINISHED:
989                 self.logger.info("Arm movement finished")
990                 return True
991         return False
992     else:
993         return True

```

1.1.4.19 **METH** `AIRBOT_PY.ARM.AIRBOTARM.MOVE_WITH_JOINT_WAYPOINTS(WAYPOINTS, BLOCKING=TRUE)`

Move the robot arm along a series of joint space waypoints. This method should run in `PLANNING_WAYPOINTS` or `PLANNING_WAYPOINTS_PATH` mode.

Parameters:

Name	Type	Description	Default
<code>waypoints</code>	<code>list[list[list[float]]]</code>	A list of joint angle sets. Each element is a list of joint positions (floats) for the 6 joints [joint1, ..., joint6].	<i>required</i>
<code>blocking</code>	<code>bool</code>	Whether to wait until execution is finished.	<code>True</code>

Returns:

Name	Type	Description
<code>bool</code>	<code>bool</code>	True if the command was accepted and (optionally) completed.

Source code in `airbot_py/arm.py`



```

995 @grpc_safe()
996 @typechecked
997 def move_with_joint_waypoints(
998     self,
999     waypoints: List[List[float]],
1000     blocking: bool = True,
1001 ) -> bool:
1002     """
1003     Move the robot arm along a series of joint space waypoints.
1004     This method should run in `PLANNING_WAYPOINTS` or `PLANNING_WAYPOINTS_PATH` mode.
1005
1006     Args:
1007         waypoints (list[list[list[float]]]): A list of joint angle sets. Each element is a
1008         list of
1009             joint positions (floats) for the 6 joints [joint1, ..., joint6].
1010         blocking (bool): Whether to wait until execution is finished.
1011
1012     Returns:
1013         bool: True if the command was accepted and (optionally) completed.
1014     """
1015
1016     def _generate_waypoints(
1017         joint_pose: List[List[List[float]]],
1018     ) -> Iterator[arm_pb2.SetStateRequest]:
1019         waypoints_obj = types_pb2.WayPoints()
1020         request = arm_pb2.SetStateRequest()
1021         now = time.time_ns()
1022         request.stamp.sec = int(now / 1e9)
1023         request.stamp.nanosec = int(now % 1e9)
1024         request.request_id = self.client_id
1025         request.component_names[:] = ["arm"]
1026
1027         for e in joint_pose:
1028             joint_states = types_pb2.JointStates()
1029             joint_states.name[:] = [
1030                 "joint1",
1031                 "joint2",
1032                 "joint3",
1033                 "joint4",
1034                 "joint5",
1035                 "joint6",
1036             ]
1037             joint_states.position[:] = e
1038             waypoints_obj.joint_states_array.extend([joint_states])
1039
1040         command = types_pb2.Command()
1041         command.waypoints.CopyFrom(waypoints_obj)
1042         request.commands.append(command)
1043         yield request
1044
1045     result_iter = self._control_stub.SetState(_generate_waypoints(waypoints))
1046     first_result = next(result_iter)
1047     if first_result.status != arm_pb2.Status.ACCEPTED:
1048         self.logger.error("Failed to move to joint position")
1049         return False
1050     if blocking:
1051         print("Waiting for arm to finish moving...")
1052         for result in result_iter:
1053             if result.status == arm_pb2.Status.FINISHED:
1054                 self.logger.info("Arm movement finished")
1055                 return True
1056         return False
1057     else:
1058         return True

```

1.1.4.20 `METH` `AIRBOT_PY.ARM.AIRBOTARM.SERVO_CART_POSE(CART_POSE)`

Send continuous cartesian pose commands to the robot. This method should run in `SERVO_CART_POSE` mode.

Source code in `airbot_py/arm.py`

```
646 @grpc_safe()
647 @typechecked
648 def servo_cart_pose(self, cart_pose: List[List[float]]) -> None:
649     """
650     Send continuous cartesian pose commands to the robot.
651     This method should run in `SERVO_CART_POSE` mode.
652     """
653     if self._servo_controller is None:
654         self._servo_controller = ServoController(
655             self.client_id, self._control_stub, self.logger
656         )
657     self._servo_controller.start_thread(self._servo_controller.servo_cmd_gen)
658     servo_cmd = types_pb2.Transform()
659     servo_cmd.translation.x = cart_pose[0][0]
660     servo_cmd.translation.y = cart_pose[0][1]
661     servo_cmd.translation.z = cart_pose[0][2]
662     servo_cmd.orientation.x = cart_pose[1][0]
663     servo_cmd.orientation.y = cart_pose[1][1]
664     servo_cmd.orientation.z = cart_pose[1][2]
665     servo_cmd.orientation.w = cart_pose[1][3]
666     self._servo_controller.append_cmd(servo_cmd)
```

1.1.4.21 `METH` `AIRBOT_PY.ARM.AIRBOTARM.SERVO_CART_TWIST(CART_TWIST)`

Send continuous cartesian twist commands to the robot. This method should run in `SERVO_CART_TWIST` mode.

Source code in `airbot_py/arm.py`

```
668 @grpc_safe()
669 @typechecked
670 def servo_cart_twist(self, cart_twist: List[List[float]]) -> None:
671     """
672     Send continuous cartesian twist commands to the robot.
673     This method should run in `SERVO_CART_TWIST` mode.
674     """
675     if self._servo_controller is None:
676         self._servo_controller = ServoController(
677             self.client_id, self._control_stub, self.logger
678         )
679     self._servo_controller.start_thread(self._servo_controller.servo_cmd_gen)
680     servo_cmd = types_pb2.Twist()
681     servo_cmd.linear.x = cart_twist[0][0]
682     servo_cmd.linear.y = cart_twist[0][1]
683     servo_cmd.linear.z = cart_twist[0][2]
684     servo_cmd.angular.x = cart_twist[1][0]
685     servo_cmd.angular.y = cart_twist[1][1]
686     servo_cmd.angular.z = cart_twist[1][2]
687     self._servo_controller.append_cmd(servo_cmd)
```

1.1.4.22 `METH` `AIRBOT_PY.ARM.AIRBOTARM.SERVO_EEF_FORCE(FORCE)`

UNIMPLEMENTED – This method has been removed and will be replaced.

.. warning:: This method is currently **UNIMPLEMENTED** and will raise a `NotImplementedError` if called.

A new force command interface will be available in **version 5.3**.
Until then, this method should not be used.

Please refer to the upcoming 5.3 API for the replacement (e.g. `send_eef_force_command()`).

Source code in `airbot_py/arm.py`

```
708 @grpc_safe()
709 @typechecked
710 def servo_eef_force(self, force: List[float]) -> None:
711     """
712     UNIMPLEMENTED – This method has been removed and will be replaced.
713
714     .. warning::
715         This method is currently UNIMPLEMENTED and will raise a NotImplementedError
716         if called.
717
718         A new force command interface will be available in version 5.3.
719         Until then, this method should not be used.
720
721         Please refer to the upcoming 5.3 API for the replacement (e.g.
722         send_eef_force_command()).
723     """
724     self.logger.warning(
725         "servo_eef_force() is UNIMPLEMENTED. \nThis method has been removed and will be
implemented in version 5.3."
    )
```

1.1.4.23 METH `AIRBOT_PY.ARM.AIRBOTARM.SERVO_EEF_POS(POS)`

Send continuous end effector width commands to the robot. This method should run in any `SERVO*` mode.

Source code in `airbot_py/arm.py`

```
689 @grpc_safe()
690 @typechecked
691 def servo_eef_pos(self, pos: Union[List[float], float]) -> None:
692     """
693     Send continuous end effector width commands to the robot.
694     This method should run in any SERVO* mode.
695     """
696     if not isinstance(pos, list):
697         pos = [pos]
698     if self._servo_controller is None:
699         self._servo_controller = ServoController(
700             self.client_id, self._control_stub, self.logger
701         )
702     self._servo_controller.start_thread(self._servo_controller.servo_cmd_gen)
703     servo_cmd = types_pb2.JointStates()
704     servo_cmd.name[:] = ["eef"]
705     servo_cmd.position[:] = pos
706     self._servo_controller.append_eef_cmd(servo_cmd)
```

1.1.4.24 METH `AIRBOT_PY.ARM.AIRBOTARM.SERVO_JOINT_POS(JOINT_POS)`

Send continuous joint position commands to the robot. This method should run in `SERVO_JOINT_POS` mode.

Source code in `airbot_py/arm.py`

```
629 @grpc_safe()
630 @typechecked
631 def servo_joint_pos(self, joint_pos: List[float]) -> None:
632     """
633     Send continuous joint position commands to the robot.
634     This method should run in `SERVO_JOINT_POS` mode.
635     """
636     if self._servo_controller is None:
637         self._servo_controller = ServoController(
638             self.client_id, self._control_stub, self.logger
639         )
640     self._servo_controller.start_thread(self._servo_controller.servo_cmd_gen)
641     servo_cmd = types_pb2.JointStates()
642     servo_cmd.name[:] = ["joint1", "joint2", "joint3", "joint4", "joint5", "joint6"]
643     servo_cmd.position[:] = joint_pos
644     self._servo_controller.append_cmd(servo_cmd)
```

1.1.4.25 **METH** `AIRBOT_PY.ARM.AIRBOTARM.SERVO_JOINT_VEL(JOINT_VEL)`

Send continuous joint velocity commands to the robot. This method should run in `SERVO_JOINT_VEL` mode.

Source code in `airbot_py/arm.py`

```
612 @grpc_safe()
613 @typechecked
614 def servo_joint_vel(self, joint_vel: List[float]) -> None:
615     """
616     Send continuous joint velocity commands to the robot.
617     This method should run in `SERVO_JOINT_VEL` mode.
618     """
619     if self._servo_controller is None:
620         self._servo_controller = ServoController(
621             self.client_id, self._control_stub, self.logger
622         )
623     self._servo_controller.start_thread(self._servo_controller.servo_cmd_gen)
624     servo_cmd = types_pb2.JointStates()
625     servo_cmd.name[:] = ["joint1", "joint2", "joint3", "joint4", "joint5", "joint6"]
626     servo_cmd.velocity[:] = joint_vel
627     self._servo_controller.append_cmd(servo_cmd)
```

1.1.4.26 **METH** `AIRBOT_PY.ARM.AIRBOTARM.SET_PARAMS(PARAMS)`

Set parameters to the driver server.

Only bool, integer, double and string types are supported.

Parameters:

Name	Type	Description	Default
<code>params</code>	<code>dict</code>	A dictionary of the parameters to set.	<i>required</i>

Return: params (dict): A dictionary of the parameters with the given keys and actual values after setting. The values are the same type as the input values.

Source code in `airbot_py/arm.py`

```
1198 @grpc_safe()
1199 @typechecked
1200 def set_params(self, params: Dict[str, Any]) -> Dict[str, Any]:
1201     """
1202     Set parameters to the driver server.
1203
1204     Only bool, integer, double and string types are supported.
1205
1206     Args:
1207         params (dict): A dictionary of the parameters to set.
1208     Return:
1209         params (dict): A dictionary of the parameters with the given keys and actual \
1210             values after setting. The values are the same type as the input values.
1211     """
1212     if self._control_stub is None:
1213         return {}
1214     if len(params) == 0:
1215         logging.error("No parameter names provided.")
1216         return {}
1217     stamp = time.time_ns()
1218     request = arm_pb2.SetParamsRequest()
1219     request.stamp.sec = int(stamp / 1e9)
1220     request.stamp.nanosec = int(stamp % 1e9)
1221     request.request_id = self.client_id
1222     for key, value in params.items():
1223         request.keys.append(key)
1224         if isinstance(value, bool):
1225             request.values.add(bool_val=value)
1226         elif isinstance(value, float):
1227             request.values.add(double_val=value)
1228         elif isinstance(value, str):
1229             request.values.add(string_val=value)
1230         elif isinstance(value, int):
1231             request.values.add(int32_val=value)
1232         else:
1233             raise TypeError(f"Unsupported type: {type(value)}")
1234     response = self._control_stub.SetParams(request)
1235     return {
1236         k: getattr(v, v.WhichOneof("union"))
1237         for k, v in zip(response.keys, response.values)
1238     }
```

1.1.4.27 METH AIRBOT_PY.ARM.AIRBOTARM.SET_SPEED_PROFILE(PROFILE)

Set the speed profile of the robot.

Three speed profiles are available:

- `SpeedProfile.DEFAULT` : Default speed profile.
- `SpeedProfile.SLOW` : Slow speed profile.
- `SpeedProfile.FAST` : Fast speed profile. **Warning:** This speed profile is experimental and may cause the robot to move too fast. It is dangerous not to make sure the robot is in a safe environment. Use at your own risk.

Parameters:

Name	Type	Description	Default
profile	SpeedProfile	The speed profile to set.	<i>required</i>

```

1240 @grpc_safe()
1241 @typechecked
1242 def set_speed_profile(self, profile: SpeedProfile):
1243     """
1244     Set the speed profile of the robot.
1245
1246     Three speed profiles are available:
1247
1248     * `SpeedProfile.DEFAULT`: Default speed profile.
1249     * `SpeedProfile.SLOW`: Slow speed profile.
1250     * `SpeedProfile.FAST`: Fast speed profile. \
1251       <span style="color: red">***Warning***: \
1252       This speed profile is experimental and may cause the robot to move too fast. \
1253       It is dangerous not to make sure the robot is in a safe environment. \
1254       Use at your own risk.</span>
1255
1256     Args:
1257         profile (SpeedProfile): The speed profile to set.
1258     """
1259     if profile not in SpeedProfile:
1260         raise ValueError(f"Invalid speed profile: {profile}")
1261     elif profile == SpeedProfile.DEFAULT:
1262         self.set_params(
1263             {
1264                 "servo_node.moveit_servo.scale.linear": 0.2,
1265                 "servo_node.moveit_servo.scale.rotational": 0.2,
1266                 "servo_node.moveit_servo.scale.joint": 0.1,
1267                 "sdk_server.max_velocity_scaling_factor": 0.5,
1268                 "sdk_server.max_acceleration_scaling_factor": 0.1,
1269             }
1270         )
1271     elif profile == SpeedProfile.SLOW:
1272         self.set_params(
1273             {
1274                 "servo_node.moveit_servo.scale.linear": 0.05,
1275                 "servo_node.moveit_servo.scale.rotational": 0.05,
1276                 "servo_node.moveit_servo.scale.joint": 0.05,
1277                 "sdk_server.max_velocity_scaling_factor": 0.1,
1278                 "sdk_server.max_acceleration_scaling_factor": 0.02,
1279             }
1280         )
1281     elif profile == SpeedProfile.FAST:
1282         self.set_params(
1283             {
1284                 "servo_node.moveit_servo.scale.linear": 10.0,
1285                 "servo_node.moveit_servo.scale.rotational": 10.0,
1286                 "servo_node.moveit_servo.scale.joint": 1.0,
1287                 "sdk_server.max_velocity_scaling_factor": 1.0,
1288                 "sdk_server.max_acceleration_scaling_factor": 0.5,
1289             }
1290         )

```

1.1.4.28 `METH AIRBOT_PY.ARM.AIRBOTARM.SWITCH_MODE(MODE)`

Switch the control mode of the robot.

The switching would not be effective if the sdk is in read-only state (when an read-write app is loaded).

Current supported mode:

- `RobotMode.PLANNING_POS` : planning mode. A single target is sent to driver server, then a execution path is planned and execute. The target could be joint positions or end cartesian pose.
- `RobotMode.SERVO_CART_TWIST` : servo mode in cartesian space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the cartesian twist commands, each of which representing the expected translational and rotational velocity of the end flange or the effecting point of the end effector.
- `RobotMode.SERVO_CART_POSE` : servo mode in cartesian space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the cartesian pose commands, each of which representing the expected cartesian pose of the end flange or the effecting point of the end effector.
- `RobotMode.SERVO_JOINT_POS` : servo mode in joint space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the joint position commands, each of which representing the expected joint position of the arm.
- `RobotMode.SERVO_JOINT_VEL` : servo mode in joint space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the joint velocity commands, each of which representing the expected joint velocity of the arm.
- `RobotMode.GRAVITY_COMP` : gravity compensation mode. In this mode, the robot would be in a free-drive state with gravity compensated. In this mode the robot would not follow any commands, but the robot could be dragged by external forces.

Parameters:

Name	Type	Description	Default
<code>mode</code>	<code>RobotMode</code>	The mode to switch to.	<i>required</i>

Returns:

Name	Type	Description
<code>result</code>	<code>bool</code>	True if the mode was switched successfully, False otherwise.

Source code in `airbot_py/arm.py`



```

536 @grpc_safe()
537 @typechecked
538 def switch_mode(self, mode: RobotMode) -> bool:
539     """
540     Switch the control mode of the robot.
541
542     The switching would not be effective if the sdk is in read-only state (when
543     an read-write app is loaded).
544
545     Current supported mode:
546
547     * `RobotMode.PLANNING_POS`: planning mode. A single target is sent to
548       driver server, then a execution path is planned and execute. The
549       target could be joint positions or end cartesian pose.
550     * `RobotMode.SERVO_CART_TWIST`: servo mode in cartesian space. Servo
551       would require continuous commands to be sent to the driver server.
552       In this mode, the robot would follow the cartesian twist commands,
553       each of which representing the expected translational and rotational
554       velocity of the end flange or the effecting point of the end effector.
555     * `RobotMode.SERVO_CART_POSE`: servo mode in cartesian space. Servo
556       would require continuous commands to be sent to the driver server.
557       In this mode, the robot would follow the cartesian pose commands,
558       each of which representing the expected cartesian pose of the end
559       flange or the effecting point of the end effector.
560     * `RobotMode.SERVO_JOINT_POS`: servo mode in joint space. Servo would
561       require continuous commands to be sent to the driver server. In this
562       mode, the robot would follow the joint position commands, each of
563       which representing the expected joint position of the arm.
564     * `RobotMode.SERVO_JOINT_VEL`: servo mode in joint space. Servo would
565       require continuous commands to be sent to the driver server. In this
566       mode, the robot would follow the joint velocity commands, each of
567       which representing the expected joint velocity of the arm.
568     * `RobotMode.GRAVITY_COMP`: gravity compensation mode. In this mode,
569       the robot would be in a free-drive state with gravity compensated.
570       In this mode the robot would not follow any commands, but the robot
571       could be dragged by external forces.
572
573     Args:
574         mode (RobotMode): The mode to switch to.
575
576     Returns:
577         result (bool): True if the mode was switched successfully, False otherwise.
578     """
579     # Per request from test team, block invalid switch mode call:
580     # RobotMode.INACTIVE
581     # RobotMode.UNDEFINED
582     if mode in (RobotMode.INACTIVE, RobotMode.UNDEFINED):
583         logging.error("Invalid mode: %s", mode.name)
584         return False
585
586     if self._control_stub is None:
587         logging.error("Not connected to the server.")
588         return False
589
590     if self._servo_controller is not None:
591         self._servo_controller.reset()
592
593     if self._mit_controller is not None:
594         self._mit_controller.reset()
595
596     now = time.time_ns()
597     request = arm_pb2.SetModeRequest()
598     request.stamp.sec = int(now / 1e9)
599     request.stamp.nanosec = int(now % 1e9)

```

```

request.request_id = self.client_id
request.component_names[:] = ["arm"]
request.modes[:] = [mode.value]
response = self._control_stub.SetMode(request)

if response.modes[0] == mode.value:
    self.logger.info("Switched to %s successfully.", mode.name)
    return True
else:
    self.logger.error("Failed to %s.", mode.name)
    return False

```

```

599
600
601
602
603
604
605
606
607
608
609
610

```

1.1.4.29 METH AIRBOT_PY.ARM.AIRBOTARM.UNLOAD_APP()

Unload the app from the driver server.

When an read-write app is unloaded, the SDK would enter read-write state, in which set methods are available. See [available states] (#airbot_py.arm--internal-states-of-the-driver)

Returns:

Name	Type	Description
result	bool	True if the app was unloaded successfully, False otherwise.

Source code in `airbot_py/arm.py`

```
1292 @grpc_safe()
1293 def unload_app(self) -> bool:
1294     """
1295     Unload the app from the driver server.
1296
1297     When an read-write app is unloaded, the SDK would enter read-write state,
1298     in which set methods are available. See [available states]
1299     (#airbot_py.arm--internal-states-of-the-driver)
1300
1301     Returns:
1302         result (bool): True if the app was unloaded successfully, False otherwise.
1303     """
1304     stamp = time.time_ns()
1305     request = utils_pb2.UnloadAppRequest()
1306     request.stamp.sec = int(stamp / 1e9)
1307     request.stamp.nanosec = int(stamp % 1e9)
1308     request.request_id = self.client_id
1309
1310     response = self._utils_stub.UnloadApp(request)
1311     return response.ok
```

1.1.5 `class` `airbot_py.arm.RobotMode`

Bases: `Enum`

Enum for the control mode of the arm.

- `PLANNING_POS` : planning mode. A single target is sent to driver server, then a execution path is planned and execute. The target could be joint positions or end cartesian pose.
- `PLANNING_WAYPOINTS_PATH` : planning mode. Perform linear interpolation between multiple waypoints in cartesian space, then execute the path without stopping.
- `PLANNING_WAYPOINTS` : planning mode. Perform local planning between adjacent waypoints, join the planned paths then execute the path. Deceleration may occur near waypoints.
- `SERVO_CART_TWIST` : servo mode in cartesian space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the cartesian twist commands, each of which representing the expected translational and rotational velocity of the end flange or the effecting point of the end effector.
- `SERVO_CART_POSE` : servo mode in cartesian space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the cartesian pose commands, each of which representing the expected cartesian pose of the end flange or the effecting point of the end effector.
- `SERVO_JOINT_POS` : servo mode in joint space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the joint position commands, each of which representing the expected joint position of the arm.
- `SERVO_JOINT_VEL` : servo mode in joint space. Servo would require continuous commands to be sent to the driver server. In this mode, the robot would follow the joint velocity commands, each of which representing the expected joint velocity of the arm.
- `MIT_INTEGRATED` : MIT integrated mode. In this mode, the robot would be in a integrated motor joint control.

- `GRAVITY_COMP` : gravity compensation mode. In this mode, the robot would be in a free-drive state with gravity compensated. In this mode the robot would not follow any commands, but the robot could be dragged by external forces.

1.1.6 `class` `airbot_py.arm.SpeedProfile`

Bases: `Enum`

Enum for speed profile of the robot.

- `DEFAULT` : The default speed profile.
- `SLOW` : The slow speed profile.
- `FAST` : The fast speed profile. **WARNING: This speed profile is experimental and may cause the robot to move too fast and cause damage to the robot or the environment. Use at your own risk .**

1.1.7 `class` `airbot_py.arm.State`

Bases: `Enum`

Enum for the state of the driver service.

- `INIT` : The driver service is initializing.
- `SHUTDOWN` : The driver service is shutting down.
- `POWERON` : The driver service is powered on and performing self-check.
- `IDLE` : The driver service is idle and waiting for external SDK commands.
- `APPLLOADING` : The driver service is loading the application.
- `APPLOADED` : The driver service has loaded the application and is ready to execute commands. When a read-write app is loaded, the robot is in read-only state and the SDK can only read the state of the robot.
- `ERROR` : The driver service is in error state.