

# CodroidPython SDK Manual

Version: 2.1.10 | Package: `codroid-robot-sdk` | Python: 3.7+

## Table of Contents

#	Chapter	Description
1	<a href="#">Quick Start</a>	Install, connect, and run your first program
2	<a href="#">Core Concepts</a>	Lifecycle, TCP model, unit conventions, error handling
3	<a href="#">CodroidSession / CodroidClient API</a>	Main client full API reference
4	<a href="#">Motion API</a>	JointPoint, CartesianPoint, MoveInstruction, MotionWaitOptions
5	<a href="#">Data Types &amp; Enums</a>	CommonResponse, CriRealTimeData, RobotFrame, GlobalVariable
6	<a href="#">CRI Real-time Data &amp; Control</a>	CriRealtimeDispatcher, TrajectoryGenerator, PacketParser
7	<a href="#">IO &amp; Registers</a>	DI/DO/AI/AO operations, register read/write
8	<a href="#">Utilities</a>	Publish/Subscribe, global variables, kinematics, ConsoleUtf8

## Requirements

Item	Requirement
Python	3.7+ (CPython / PyPy, 3.7 ~ 3.14)
OS	Linux, Windows, macOS
Runtime deps	None (optional <code>colorama</code> for colored terminal output)

## Install

```
pip install codroid-robot-sdk
```

Optional colored terminal output:

```
pip install "codroid-robot-sdk[color]"
```

## Verify Installation

```
python3 -c "import codroid; print(codroid.__version__)"
```

## API Naming Convention

All public methods use **PascalCase** (consistent with C# / C++ SDKs).

```
# Correct
robot.ConnectRemoteAndSwitchOn()
di = robot.GetDi(0)

# Wrong – snake_case removed in 2.1.1
robot.connect_remote_and_switch_on() # does not exist
```

## Unit Conventions

Layer	Linear	Angular
SDK public API	mm	deg (degrees)
TCP JSON protocol	mm	deg
CRI UDP binary (wire)	m	rad (radians)
<code>CriRealTimeData</code> (parsed)	mm	deg

`CriRealtimePacketParser.parse()` and `CriRealtimeDispatcher` (`convert_to_si=True`) automatically handle m ↔ mm and rad ↔ deg conversion.

# Quick Start

---

## Install

### Via pip

```
pip install codroid-robot-sdk
```

### From Source

```
git clone https://github.com/guybod/CodroidSDK.git
cd CodroidSDK/CodroidPython
pip install -e .
```

---

## Minimal Example

Connect to the controller, enter remote mode, and enable.

```
from codroid import CodroidControlInterface, InitConsoleUtf8

InitConsoleUtf8() # Windows cmd UTF-8 fix; no-op on Linux

ROBOT_IP = "192.168.1.136"

with CodroidControlInterface(host=ROBOT_IP) as robot:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
```

Run:

```
python3 demo.py
```

---

## Complete Workflow Example

```
from codroid import (
    CodroidControlInterface,
    JointPoint,
    CartesianPoint,
    MoveInstruction,
    InitConsoleUtf8,
)

InitConsoleUtf8()

ROBOT_IP = "192.168.1.136"

with CodroidControlInterface(host=ROBOT_IP) as robot:
```

```
# 1. Connect and power on
robot.EnterRemoteModeViaAuto()
robot.SwitchOn()

# 2. IO operations
di0 = robot.GetDi(0)
robot.SetDo(10, di0)

# 3. Registers
reg_val = robot.GetRegisterValue(0)
robot.SetRegisterValue(0, reg_val + 1)

# 4. Joint motion
robot.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acceleration=100)

# 5. Linear motion
robot.MovL(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]),
           speed=150, acceleration=500)
```

---

## Using CodroidClient

`CodroidClient` inherits from `CodroidSession`, uses a background thread for receiving data, and supports publish/subscribe event dispatch.

```
from codroid import CodroidClient, InitConsoleUtf8

InitConsoleUtf8()

with CodroidClient(host="192.168.1.136") as robot:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
```

---

## Running Examples

```
# Basic usage
PYTHONPATH=src python examples/01_basic_usage.py --robot 192.168.8.136

# Motion examples
PYTHONPATH=src python examples/08_move.py --robot 192.168.8.136

# Blocking motion
PYTHONPATH=src python examples/15_sync_motion.py --robot 192.168.8.136

# Robot settings
PYTHONPATH=src python examples/14_robot_parameters.py --robot 192.168.8.136
```

---

## Error Handling

All TCP commands throw exceptions on failure:

Exception	Condition
<code>CodroidError</code>	Base exception
<code>CodroidCommandException</code>	Controller returned <code>err</code> field
<code>CodroidNetworkError</code>	TCP connection or communication failure
<code>CodroidTimeoutError</code>	Operation timeout

```
from codroid import CodroidControlInterface, CodroidError, CodroidTimeoutError

try:
    with CodroidControlInterface(host="192.168.1.136") as robot:
        robot.EnterRemoteModeViaAuto()
        robot.SwitchOn()
        robot.MovJ([0, 0, 90, 0, 90, 0], speed=40, acceleration=100)
except CodroidTimeoutError:
    print("Operation timed out")
except CodroidError as e:
    print(f"SDK error: {e}")
```

---

## Windows Console UTF-8

When running examples with Chinese characters in `cmd` (not Windows Terminal), call at entry:

```
from codroid import InitConsoleUtf8

InitConsoleUtf8()
```

All `examples/*.py` call this at the top of `if __name__ == "__main__"`. No-op on Linux / macOS.

# Core Concepts

## Client Lifecycle

Typical lifecycle for `CodroidSession` (alias `CodroidControlInterface`) and `CodroidClient`:

```
from codroid import CodroidClient

# Option 1: with statement (recommended)
with CodroidClient(host="192.168.1.136") as robot:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
    # ... use API ...
# Disconnect() called automatically

# Option 2: manual management
robot = CodroidClient(host="192.168.1.136")
robot.Connect()
try:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
    # ... use API ...
finally:
    robot.Disconnect()
```

## CodroidSession vs CodroidClient

Feature	CodroidSession	CodroidClient
Transport	<code>JsonStreamClient</code> (sync blocking)	<code>TransportClient</code> (background thread)
Request/Response matching	Sync send-receive	Async ID matching
Publish/Subscribe	Not supported	Supported
Use case	Simple scripts, one-shot ops	Continuous receive, event-driven

## TCP Command Model

SDK communicates with the controller via TCP JSON. Each command flow:

1. SDK assigns auto-incrementing `id`
2. Sends request: `{"id": N, "ty": "command/path", "db": {...}}`
3. Controller responds: `{"id": N, "ty": "...", "db": {...}, "err": ...}`
4. SDK matches request and response by `id`

```
# SDK handles id assignment and matching internally
response = robot._send_command("Robot/switchOn", "")
# response = CommonResponse(id=1, ty="Robot/switchOn", db=None, err=None)
```

## CommonResponse

All TCP commands return `CommonResponse`:

```
@dataclass
class CommonResponse:
    id: Union[int, str]    # Request ID
    ty: str               # Response type
    db: Optional[Any]     # Response data
    err: Optional[Any]    # Error info (None means success)

    @property
    def is_success(self) -> bool:
        return self.err is None
```

## Error Handling

SDK defines four exception types:

Exception	Trigger
<code>CodroidError</code>	Base exception; parameter validation failure, illegal operations
<code>CodroidCommandException</code>	Controller returned <code>err</code> field (protocol-level error)
<code>CodroidNetworkError</code>	TCP connection failure, communication interruption
<code>CodroidTimeoutError</code>	Operation timeout (connection, CRI wait, blocking motion)

```
from codroid import (
    CodroidError,
    CodroidCommandException,
    CodroidNetworkError,
    CodroidTimeoutError,
)

try:
    with CodroidClient(host="192.168.1.136") as robot:
        robot.ConnectRemoteAndSwitchOn()
except CodroidNetworkError:
    print("Cannot connect to controller")
except CodroidTimeoutError:
    print("Connection timed out")
except CodroidCommandException as e:
    print(f"Controller error: {e}")
except CodroidError as e:
```

```
print(f"SDK error: {e}")
```

---

## Thread Safety

- `CriData` property returns the current cached CRI snapshot, readable from any thread.
  - TCP methods (`GetDi`, `MovJ`, etc.) can be called from any thread, but concurrent calls on the same client instance are not supported.
  - `CriRealtimeDispatcher`'s `SendCommand` / `SendTrajectory` are thread-safe (UDP is stateless).
- 

## Publish / Subscribe

`CodroidClient` supports subscribing to controller push event topics:

```
from codroid import CodroidClient, PublishTopics

def on_robot_status(notification):
    print(f"Received {notification.ty}: {notification.db}")

with CodroidClient(host="192.168.1.136") as robot:
    sub = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_robot_status)
    # ... run ...
    sub.dispose() # Unsubscribe
```

Available topics in `PublishTopics`:

- `PROJECT_STATE` — Project state change
- `VAR_UPDATE` — Variable update
- `ROBOT_STATUS` — Robot status
- `ROBOT_POSTURE` — Robot posture
- `ROBOT_COORDINATE` — Robot coordinate
- `LOG` — Log
- `ERROR` — Error



# CodroidSession / CodroidClient API Reference

`CodroidSession` (alias `CodroidControlInterface`) is the main session class encapsulating all TCP commands. `CodroidClient` inherits it and replaces the transport layer, adding publish/subscribe capabilities.

## Constructor

### CodroidSession

```
CodroidSession(  
    host: str = "192.168.1.136",  
    port: int = 9001,  
    local_ip: str = "192.168.1.150",  
    udp_port: int = 10086,  
)
```

Parameter	Type	Default	Description
host	str	"192.168.1.136"	Controller IP address
port	int	9001	TCP port
local_ip	str	"192.168.1.150"	Local IP (for CRI UDP push)
udp_port	int	10086	Local UDP listening port

### CodroidClient

```
CodroidClient(  
    host: str = "192.168.1.136",  
    port: int = 9001,  
    local_ip: str = "192.168.1.150",  
    udp_port: int = 10086,  
    timeout: float = 10.0,  
)
```

Additional parameter:

Parameter	Type	Default	Description
timeout	float	10.0	TCP request timeout (seconds)

# Properties

## CriData

```
@property
def CriData(self) -> Optional[CriRealTimeData]
```

Latest CRI real-time data snapshot. Requires calling `StartListenUdp()` or `StartCriDataPush()` first.

```
robot.StartListenUdp()
robot.WaitForCriData()
data = robot.CriData
print(f"Joint angles: {data.joint_position}")
print(f"TCP pose: {data.tcp_pose}")
print(f"Is moving: {data.status.is_moving}")
```

---

## Connection Management

### Connect

```
def Connect(self) -> CodroidSession
```

Establish TCP connection. Returns self for chaining. Called automatically with `with` statement.

### Disconnect

```
def Disconnect(self) -> None
```

Disconnect TCP and stop CRI receive. Called automatically with `with` statement.

---

## Convenience Connect

### ConnectRemoteAndSwitchOn

```
def ConnectRemoteAndSwitchOn(self) -> CommonResponse
```

Combined operation: `EnterRemoteModeViaAuto()` → `SwitchOn()`. Must `Connect()` first or be inside `with` block.

```
with CodroidClient(host="192.168.1.136") as robot:
    robot.ConnectRemoteAndSwitchOn()
```

## EnterRemoteModeViaAuto

```
def EnterRemoteModeViaAuto(self) -> CommonResponse
```

Calls `ToAuto()` then `ToRemote()`.

## EnterManualModeViaAuto

```
def EnterManualModeViaAuto(self) -> CommonResponse
```

Calls `ToAuto()` then `ToManual()`.

---

# Mode Switching

## SwitchOn / SwitchOff

```
def SwitchOn(self) -> CommonResponse  
def SwitchOff(self) -> CommonResponse
```

Enable / disable the robot.

## ToRemote / ToManual / ToAuto

```
def ToRemote(self) -> CommonResponse  
def ToManual(self) -> CommonResponse  
def ToAuto(self) -> CommonResponse
```

Switch to remote / manual / auto mode.

## ToSimulation / ToActual

```
def ToSimulation(self) -> CommonResponse  
def ToActual(self) -> CommonResponse
```

Switch to simulation / actual mode.

## StartDrag / StopDrag

```
def StartDrag(self) -> CommonResponse  
def StopDrag(self) -> CommonResponse
```

Enter / exit drag-and-teach mode. Only available in remote or manual mode.

## ClearSystemError

```
def ClearSystemError(self) -> CommonResponse
```

Clear system errors.

## Non-Blocking Motion Commands

### MovJ

```
def MovJ(  
    self,  
    target: Union[JointPoint, CartesianPoint, MovePoint],  
    speed: float,  
    acceleration: float,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

Joint interpolation motion. Target can be `JointPoint` (sends jp) or `CartesianPoint` (sends cp+rj).

Parameter	Type	Description
target	JointPoint / CartesianPoint	Motion target
speed	float	Speed
acceleration	float	Acceleration
blend	float	Blend radius, default 0 (exact arrival)
coor	Sequence[float]	User coordinate [x, y, z, a, b, c]
tool	Sequence[float]	Tool coordinate [x, y, z, a, b, c]

```
robot.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acceleration=100)  
robot.MovJ(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]), speed=40, acceleration=100)
```

## MovL

```
def MovL(  
    self,  
    target: Union[CartesianPoint, JointPoint, MovePoint],  
    speed: float,  
    acceleration: float,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

Linear interpolation motion. Target can be `CartesianPoint` or `JointPoint`.

## MovC

```
def MovC(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    speed: float,  
    acceleration: float,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

Circular arc motion. Both middle and target are `CartesianPoint`.

## MovCircle

```
def MovCircle(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    circle_num: int,  
    speed: float,  
    acceleration: float,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

Full circle motion. `circle_num` is the number of revolutions.

## Move

```
def Move(  
    self,  
    path: Union[MotionPath, List[MoveInstruction], List[Dict[str, Any]]],  
) -> CommonResponse
```

Multi-segment path execution. `List[MoveInstruction]` is recommended.

```
path = [  
    MoveInstruction.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acc=100),  
    MoveInstruction.MovL(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]), speed=150,  
acc=500),  
    MoveInstruction.MovL(JointPoint.Degrees([0, 0, 0, 0, 0, 0]), speed=150, acc=500),  
]  
robot.Move(path)
```

## Blocking Motion Commands

`*Sync` methods send the motion command then automatically poll CRI data until the robot stops. CRI data push must be started first.

**2.1.10 behavior change:** Completion is determined solely by the CRI `InMotion` flag (was moving + consecutive `settled_samples` stable reads). Joint/TCP position vs target is **no longer checked**.

`MotionWaitOptions` tolerance fields are deprecated.

### MovJSync

```
def MovJSync(  
    self,  
    target: Union[JointPoint, CartesianPoint],  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

Blocking joint motion. Returns `True` on reaching the target.

```
robot.StartListenUdp()  
robot.WaitForCriData()  
robot.MovJSync(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acceleration=100)
```

## MovLSync

```
def MovLSync(  
    self,  
    target: Union[CartesianPoint, JointPoint],  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

Blocking linear motion.

## MovCSync

```
def MovCSync(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

Blocking circular arc motion.

## MovCircleSync

```
def MovCircleSync(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    circle_num: int,  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

Blocking full circle motion.

## MoveSync

```
def MoveSync(  
    self,  
    path: Union[MotionPath, List[MoveInstruction], List[Dict[str, Any]]],  
    wait: Optional[MotionWaitOptions] = None,  
) -> bool
```

Blocking path execution. Waits for CRI to confirm the last segment reached the target.

## MotionWaitOptions

Controls blocking motion wait behavior.

**2.1.10 change:** Completion is determined solely by the CRI `InMotion` flag. Tolerance fields are deprecated.

```
@dataclass  
class MotionWaitOptions:  
    timeout: float = 60.0           # Overall timeout (seconds)  
    poll_interval: float = 0.05    # Poll interval (seconds)  
    cri_stale_timeout: float = 0.5 # CRI data stale threshold (seconds)  
    settled_samples: int = 3        # Consecutive settled sample count
```

```
from codroid import MotionWaitOptions  
  
opts = MotionWaitOptions(timeout=30.0)  
robot.MovJSync(JointPoint.Degrees([0, 0, 90, 0, 90, 0]),  
               speed=40, acceleration=100, wait=opts)
```

---

## Motion Control

### PauseRobotMotion / ResumeRobotMotion

```
def PauseRobotMotion(self) -> CommonResponse  
def ResumeRobotMotion(self) -> CommonResponse
```

Pause / resume current motion.

### StopRobotMove

```
def StopRobotMove(self) -> CommonResponse
```

Stop current motion.

---



# MoveTo Commands

## MoveTo

```
def MoveTo(  
    self,  
    move_type: MoveToType,  
    target: Optional[MoveToTarget] = None,  
) -> CommonResponse
```

Move to a preset position. `MoveToType` enum values:

Value	Description
<code>STOP</code> (-1)	Stop MoveTo
<code>HOME</code> (0)	Home position
<code>SAFE</code> (1)	Safe position
<code>CANDLE</code> (2)	Candle position
<code>PACK</code> (3)	Pack position
<code>JOINT</code> (4)	Joint planned to target
<code>LINEAR</code> (5)	Linear planned to target
<code>RESUME</code> (6)	Program resume point

```
from codroid import MoveToType  
  
robot.MoveTo(MoveToType.HOME) # Go to Home  
robot.MoveTo(MoveToType.SAFE) # Go to Safe
```

Must call `MoveToHeartbeat()` every 0.5s during MoveTo motion.

## MoveToHeartbeat

```
def MoveToHeartbeat(self) -> CommonResponse
```

MoveTo heartbeat. Must be called every 0.5s during MoveTo motion.

## StopMoveTo

```
def StopMoveTo(self) -> CommonResponse
```

Stop current MoveTo motion.

---

# Jog Commands

## StartJog

```
def StartJog(  
    self,  
    mode: JogMode,  
    index: int,  
    speed: float,  
    coor_type: JogCoorType = JogCoorType.USER,  
    coor_id: int = 1,  
) -> CommonResponse
```

Start jogging.

Parameter	Type	Description
mode	JogMode	JOINT (1) joint jog / LINEAR (2) linear jog
index	int	Joint number (1-6) or linear axis (1-6 for xyzabc)
speed	float	Speed (-1.0 ~ 1.0)
coor_type	JogCoorType	USER (0) / TOOL (1)
coor_id	int	User coordinate ID

```
from codroid import JogMode  
  
robot.StartJog(JogMode.JOINT, index=1, speed=0.5) # Joint 1 positive jog
```

## StopJog

```
def StopJog(self) -> CommonResponse
```

Stop jogging.

## JogHeartbeat

```
def JogHeartbeat(self) -> CommonResponse
```

Jog heartbeat. Must be called every 0.5s during jogging.

---

# Speed Rate

## SetManualMoveRate / SetAutoMoveRate

```
def SetManualMoveRate(self, rate: int) -> CommonResponse
def SetAutoMoveRate(self, rate: int) -> CommonResponse
```

Set manual / auto move rate. `rate` range 1~100.

---

## CRI Data Push

### StartListenUdp

```
def StartListenUdp(self) -> None
```

One-click CRI data receive setup: stop old push → start local UDP listener → notify controller to start push.

```
robot.StartListenUdp()
robot.WaitForCriData()
data = robot.CriData
```

### WaitForCriData

```
def WaitForCriData(self, timeout: float = 5.0) -> CriRealTimeData
```

Wait for the first CRI data packet. Throws `CodroidTimeoutError` on timeout.

### StartCriDataPush

```
def StartCriDataPush(
    self,
    ip: str,
    port: int,
    duration: int = 100,
    high_precision: bool = True,
    mask: int = 0xFFFF,
) -> CommonResponse
```

Manually start CRI UDP push. `port` range 10000–65534. `duration` is push period (ms).

### StopCriDataPush

```
def StopCriDataPush(self, ip: Optional[str] = None, port: Optional[int] = None) ->
CommonResponse
```

Stop CRI push.

## StartCriControl

```
def StartCriControl(  
    self,  
    filter_type: CriFilterType = CriFilterType.NONE,  
    duration: int = 1,  
    start_buffer: int = 3,  
) -> CommonResponse
```

Start CRI real-time control mode. `duration` is command interval (ms, 1–16, must divide 1000).

## StopCriControl

```
def StopCriControl(self) -> CommonResponse
```

Stop CRI real-time control.

---

## IO Operations

### GetDi / GetDo / GetAi / GetAo

```
def GetDi(self, port: int) -> int      # Digital input, port 0~15, returns 0 or 1  
def GetDo(self, port: int) -> int      # Digital output, port 0~15, returns 0 or 1  
def GetAi(self, port: int) -> float    # Analog input, port 0~3  
def GetAo(self, port: int) -> float    # Analog output, port 0~3
```

```
di0 = robot.GetDi(0)  
do5 = robot.GetDo(5)  
ai0 = robot.GetAi(0)
```

### SetDo / SetAo

```
def SetDo(self, port: int, value: int) -> CommonResponse    # value: 0 or 1  
def SetAo(self, port: int, value: float) -> CommonResponse
```

```
robot.SetDo(10, 1)  
robot.SetAo(0, 3.14)
```

### GetIoValues

```
def GetIoValues(self, io_requests: List[Dict[str, Any]]) -> CommonResponse
```

Batch read IO. `io_requests` format: `[{"type": "DI", "port": 0}, ...]`.

---

# Register Operations

## GetRegisterValue / GetRegisterValues

```
def GetRegisterValue(self, address: int) -> Any
def GetRegisterValues(self, addresses: List[int]) -> CommonResponse
```

```
val = robot.GetRegisterValue(0)
vals = robot.GetRegisterValues([0, 1, 2])
```

## SetRegisterValue

```
def SetRegisterValue(self, address: int, value: Any) -> CommonResponse
```

```
robot.SetRegisterValue(0, 42)
```

## SetExtendArrayType / RemoveExtendArray

```
def SetExtendArrayType(self, index: int, data_type: ExtendArrayType) -> CommonResponse
def RemoveExtendArray(self, index: int) -> CommonResponse
```

Set / remove extended array data type. `index` range 0~999.

---

# Robot Settings

## GetRobotParameters

```
def GetRobotParameters(self) -> RobotParameters
```

Get settings interface parameter snapshot. Returns `RobotParameters` containing tool frame table, payload frame table, coordinate frame table, and default IDs.

```
params = robot.GetRobotParameters()
print(f"Default tool: {params.default_tool_id}")
print(f"Default payload: {params.default_payload_id}")
for frame in params.tool:
    print(f"  Tool[{frame.id}]: x={frame.x}, y={frame.y}, z={frame.z}")
```

## SetToolFrame / SaveToolFrames

```
def SetToolFrame(self, frame_id: int, frame) -> CommonResponse
def SaveToolFrames(self, frames) -> CommonResponse
```

`SetToolFrame` modifies a single tool frame (read-modify-write, `frame_id` 1~15 only). `SaveToolFrames` replaces the full table (16 items, id=0 must be all zeros).

```
from codroid import RobotFrame

robot.SetToolFrame(1, RobotFrame(id=1, x=100, y=0, z=50, a=0, b=0, c=0))
```

## SetPayloadFrame / SavePayloadFrames

```
def SetPayloadFrame(self, frame_id: int, frame) -> CommonResponse
def SavePayloadFrames(self, frames) -> CommonResponse
```

`SetPayloadFrame` modifies a single payload frame ( `frame_id` 1~15 only). `SavePayloadFrames` replaces the full table.

```
from codroid import RobotPayloadFrame

robot.SetPayloadFrame(1, RobotPayloadFrame(id=1, m=2.5, mx=0, my=0, mz=50))
```

## SetUserCoordinateFrame / SaveUserCoordinateFrames

```
def SetUserCoordinateFrame(self, frame_id: int, frame) -> CommonResponse
def SaveUserCoordinateFrames(self, frames) -> CommonResponse
```

Modify single / replace full user coordinate frame table.

## SetDefaultToolId / SetDefaultPayloadId / SetDefaultUserCoordinateId

```
def SetDefaultToolId(self, tool_id: int) -> CommonResponse
def SetDefaultPayloadId(self, payload_id: int) -> CommonResponse
def SetDefaultUserCoordinateId(self, coordinate_id: int) -> CommonResponse
```

Set default tool / payload / user coordinate ID. Range 0~15.

## SetCollisionSensitivity

```
def SetCollisionSensitivity(self, sensitivity: int) -> CommonResponse
```

Set collision detection sensitivity. Range 0~100. Firmware 2.3.2.10+ only.

## SetPayload

```
def SetPayload(self, payload_id: int) -> CommonResponse
```

Set current payload ID. Firmware 2.3.2.10+ only.

---

# Project & Script

## RunScript

```
def RunScript(  
    self,  
    main_script: str,  
    sub_threads: Optional[Dict[str, str]] = None,  
    sub_programs: Optional[Dict[str, str]] = None,  
    interrupts: Optional[Dict[str, str]] = None,  
    vars: Optional[Dict[str, Any]] = None,  
) -> CommonResponse
```

Run Lua script.

```
robot.RunScript('print("hello")', vars={"speed": 100})
```

## Run / RunByIndex / RunStep

```
def Run(self, project_id: str) -> CommonResponse  
def RunByIndex(self, index: int) -> CommonResponse  
def RunStep(self, project_id: str) -> CommonResponse
```

Run / single-step a project.

## PauseProject / ResumeProject / StopProject

```
def PauseProject(self) -> CommonResponse  
def ResumeProject(self) -> CommonResponse  
def StopProject(self) -> CommonResponse
```

Pause / resume / stop a project.

## EnterRemoteScriptMode

```
def EnterRemoteScriptMode(self) -> CommonResponse
```

Enter remote script mode.

## SetStartLine / ClearStartLine

```
def SetStartLine(self, line: int) -> CommonResponse  
def ClearStartLine(self) -> CommonResponse
```

Set / clear start line.

---

# RS485 Communication

## Rs485Init

```
def Rs485Init(  
    self,  
    baudrate: Union[RS485BaudRate, int],  
    stop_bit: RS485StopBits = RS485StopBits.ONE,  
    parity: RS485Parity = RS485Parity.NONE,  
) -> CommonResponse
```

Initialize end-effector RS485.

```
from codroid import RS485BaudRate  
  
robot.Rs485Init(RS485BaudRate.B115200)
```

## Rs485Flush

```
def Rs485Flush(self) -> CommonResponse
```

Flush RS485 read buffer.

## Rs485Read

```
def Rs485Read(self, length: int, timeout: int = 3000) -> CommonResponse
```

Read RS485 data. `length` max 128 bytes, `timeout` max 3000ms.

## Rs485Write

```
def Rs485Write(self, data: Union[List[int], bytes]) -> CommonResponse
```

Write RS485 data. Max 127 bytes.

---

# Kinematics

## AposToCpos

```
def AposToCpos(  
    self,  
    jp: Sequence[float],  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
    ep: Sequence[float] = [],  
) -> CommonResponse
```

Forward kinematics (joint → Cartesian). `jp` is 6 joint angles (degrees).



## CposToApos

```
def CposToApos(  
    self,  
    cp: Sequence[float],  
    rj: Optional[Sequence[float]] = None,  
    ep: Sequence[float] = [],  
) -> CommonResponse
```

Inverse kinematics (Cartesian  $\rightarrow$  joint). `cp` is `[x, y, z, a, b, c]`, `rj` is reference joint angles (default `[20, 20, 20, 20, 20, 20]`).

## CalculateRelativePose

```
def CalculateRelativePose(  
    self,  
    pos: Sequence[float],  
    offset: Sequence[float],  
    coor_type: CoordinateType = CoordinateType.TOOL,  
    pos_coor: Optional[Sequence[float]] = None,  
    coor: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

Cartesian coordinate offset calculation.

---

## Publish / Subscribe (CodroidClient only)

### SubscribePublishTopic

```
def SubscribePublishTopic(  
    self,  
    topic_ty: str,  
    handler: Callable[[PublishNotification], None],  
    tc_milliseconds: int = 100,  
) -> PublishTopicSubscription
```

Subscribe to controller push topic. Returns `PublishTopicSubscription`, call `.dispose()` to unsubscribe.

```
from codroid import CodroidClient, PublishTopics  
  
def on_status(notification):  
    print(f"{notification.ty}: {notification.db}")  
  
with CodroidClient(host="192.168.1.136") as robot:  
    sub = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_status)  
    # ... run ...  
    sub.dispose()
```

---

## Debug

```
robot.debug = True # Print sent/received raw JSON
```

# Motion API Reference

---

## Motion Target Types

### JointPoint

```
@dataclass
class JointPoint:
    jp: List[float] # 6 joint angles (degrees)
```

Joint target. Used at the business layer to declare "this is a 6-axis joint angle", avoiding confusion with TCP pose.

#### Factory Method

```
JointPoint.Degrees(joints_deg: Sequence[float]) -> JointPoint
```

Construct from six joint angles (degrees).

```
jp = JointPoint.Degrees([0, 0, 90, 0, 90, 0])
```

---

### CartesianPoint

```
@dataclass
class CartesianPoint:
    cp: List[float] # TCP pose [x,y,z,rx,ry,rz] (mm + degrees)
    rj: Optional[List[float]] = None # IK reference joint angles (degrees)
```

Cartesian target. `cp` is required; `rj` is optional (defaults to controller default reference joints when packing).

#### Factory Methods

```
CartesianPoint.MmDeg(pose_mm_deg: Sequence[float]) -> CartesianPoint
```

TCP pose `[x,y,z,rx,ry,rz]` (mm + degrees), no reference joints.

```
CartesianPoint.MmDegWithRef(
    pose_mm_deg: Sequence[float],
    ref_joints_deg: Sequence[float],
) -> CartesianPoint
```

TCP pose + IK reference joints (degrees). Recommended when using `movJ/movL` to TCP and caring about orientation solution.

```
cp = CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90])
cp_with_ref = CartesianPoint.MmDegWithRef(
    [400, 200, 500, 180, 0, 90],
    [0, 0, 90, 0, 90, 0],
)
```

---

## MovePoint

```
@dataclass
class MovePoint:
    jp: Optional[Sequence[float]] = None
    cp: Optional[Sequence[float]] = None
    rj: Optional[Sequence[float]] = None
    ep: Optional[Sequence[float]] = None
```

General motion point definition. Usually not used directly; constructed via `JointPoint` / `CartesianPoint`.

### Factory Methods

```
MovePoint.FromJoint(joint: JointPoint) -> MovePoint
MovePoint.FromCartesian(cart: CartesianPoint) -> MovePoint
```

---

## Motion Instructions

### MoveInstruction

```
@dataclass
class MoveInstruction:
    motion_type: MotionType
    target: MovePoint
    speed: float
    acc: float
    blend: Optional[float] = None
    relative_blend: Optional[float] = None
    middle: Optional[MovePoint] = None
    circle_num: Optional[int] = None
    coor: Optional[Sequence[float]] = None
    tool: Optional[Sequence[float]] = None
```

Single-segment `Robot/move` instruction. Use class methods to construct; do not manually write `type` + bare `MovePoint`.

## Factory Methods

### MoveInstruction.MovJ

```
MoveInstruction.MovJ(  
  target: Union[JointPoint, CartesianPoint],  
  speed: float,  
  acc: float,  
  blend: Optional[float] = None,  
  relative_blend: Optional[float] = None,  
  coor: Optional[Sequence[float]] = None,  
  tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

Joint motion movJ. Target can be joint or TCP.

### MoveInstruction.MovL

```
MoveInstruction.MovL(  
  target: Union[CartesianPoint, JointPoint],  
  speed: float,  
  acc: float,  
  blend: Optional[float] = None,  
  relative_blend: Optional[float] = None,  
  coor: Optional[Sequence[float]] = None,  
  tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

Linear motion movL. Target can be TCP or joint.

### MoveInstruction.MovC

```
MoveInstruction.MovC(  
  middle: CartesianPoint,  
  target: CartesianPoint,  
  speed: float,  
  acc: float,  
  blend: Optional[float] = None,  
  relative_blend: Optional[float] = None,  
  coor: Optional[Sequence[float]] = None,  
  tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

Circular arc motion movC. Both middle and target are TCP.

## MoveInstruction.MovCircle

```
MoveInstruction.MovCircle(  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    circle_num: int,  
    speed: float,  
    acc: float,  
    blend: Optional[float] = None,  
    relative_blend: Optional[float] = None,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

Full circle motion movCircle.

---

## MotionType

```
class MotionType(str, Enum):  
    MOVJ = "movJ"  
    MOVL = "movL"  
    MOVC = "movC"  
    MOVIRCLE = "movCircle"
```

---

## MoveTo Targets

### MoveToTarget

```
@dataclass  
class MoveToTarget:  
    cp: Optional[Sequence[float]] = None  
    jp: Optional[Sequence[float]] = None  
    ep: Sequence[float] = field(default_factory=list)
```

MoveTo-specific target structure.

### Factory Methods

```
MoveToTarget.Joint(joint: JointPoint) -> MoveToTarget  
MoveToTarget.Cartesian(cart: CartesianPoint) -> MoveToTarget
```

## MoveToType

```
class MoveToType(IntEnum):
    STOP = -1      # Stop MoveTo
    HOME = 0       # Home position
    SAFE = 1       # Safe position
    CANDLE = 2     # Candle position
    PACK = 3       # Pack position
    JOINT = 4      # Joint planned
    LINEAR = 5     # Linear planned
    RESUME = 6     # Program resume point
```

---

## Jog Parameters

### JogMode

```
class JogMode(IntEnum):
    JOINT = 1      # Joint jog
    LINEAR = 2     # Linear jog
```

### JogCoorType

```
class JogCoorType(IntEnum):
    USER = 0      # User coordinate
    TOOL = 1       # Tool coordinate
```

---

## MotionPath (Legacy API)

```
class MotionPath:
    def add(self, instruction: MoveInstruction) -> MotionPath
    def MovJ(self, target, speed, acc, blend=None) -> MotionPath
    def MovL(self, target, speed, acc, blend=None) -> MotionPath
    def MovC(self, target, middle, speed, acc, blend=None) -> MotionPath
    def clear(self) -> None
    def get_commands(self) -> List[Dict[str, Any]]
```

Motion path builder. New code should prefer `List[MoveInstruction] + Move()`.

---

## Complete Multi-Segment Path Example

```
from codroid import (
    CodroidClient,
    JointPoint,
    CartesianPoint,
    MoveInstruction,
    MotionWaitOptions,
```

```

    InitConsoleUtf8,
)

InitConsoleUtf8()

with CodroidClient(host="192.168.8.136") as robot:
    robot.ConnectRemoteAndSwitchOn()

    # Four-combination path
    path = [
        MoveInstruction.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]),
                             speed=40, acc=100),
        MoveInstruction.MovJ(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]),
                             speed=40, acc=100),
        MoveInstruction.MovL(CartesianPoint.MmDeg([400, -200, 500, 180, 0, 90]),
                             speed=150, acc=500),
        MoveInstruction.MovL(JointPoint.Degrees([0, 0, 0, 0, 0, 0]),
                             speed=150, acc=500),
    ]
    robot.Move(path)

    # Blocking path execution
    robot.StartListenUdp()
    robot.WaitForCriData()
    robot.MoveSync(path, wait=MotionWaitOptions(timeout=120.0))

```

## Motion Parameter Notes

### speed / acceleration

- `speed`: Speed value. Exact unit depends on motion type and controller configuration.
- `acceleration`: Acceleration value.

### blend

Blend radius. Default `0.0` (exact arrival at target). When set to a value  $> 0$ , the robot smoothly transitions near the target without stopping.

### coord / tool

Optional user coordinate and tool coordinate, format `[x, y, z, a, b, c]` (mm + degrees).



# Data Types & Enums

---

## Communication Models

### CommonResponse

```
@dataclass
class CommonResponse:
    id: Union[int, str]
    ty: str
    db: Optional[Any] = None
    err: Optional[Any] = None

    @property
    def is_success(self) -> bool:
        return self.err is None
```

TCP JSON common response. All TCP commands return this type.

Field	Type	Description
id	int / str	Request ID
ty	str	Response type
db	Any	Response data
err	Any	Error info (None means success)

Legacy alias: `CodroidResponse = CommonResponse`

---

### CodroidRequest

```
@dataclass
class CodroidRequest:
    id: Union[int, str]
    ty: str
    db: Optional[Any] = None
```

SDK internal request structure.

---

# CRI Real-time Data

## CriRealTimeData

```
@dataclass
class CriRealTimeData:
    timestamp: int = 0
    status: CriStatus = field(default_factory=CriStatus)
    joint_pos: List[float] = field(default_factory=list)
    joint_vel: List[float] = field(default_factory=list)
    cartesian_pos: List[float] = field(default_factory=list)
    cartesian_vel: List[float] = field(default_factory=list)
    tcp_speed: float = 0.0
    joint_torque: List[float] = field(default_factory=list)
    external_torque: List[float] = field(default_factory=list)
    extra_axis_pos: List[float] = field(default_factory=list)
```

CRI parsed real-time data. All values converted to mm + degrees.

Field	Property Alias	Type	Unit	Description
timestamp	timestamp_ms	int	ms	Timestamp
status	—	CriStatus	—	Status bits
joint_pos	joint_position	List[float]	deg	6 joint angles
joint_vel	joint_velocity	List[float]	deg/s	6 joint velocities
cartesian_pos	tcp_pose	List[float]	mm, deg	TCP pose [x,y,z,rx,ry,rz]
cartesian_vel	tcp_velocity	List[float]	mm/s, deg/s	TCP velocity
tcp_speed	tcp_linear_velocity	float	mm/s	TCP linear speed
joint_torque	joint_output_torque	List[float]	Nm	Joint output torque
external_torque	joint_external_force	List[float]	Nm	External torque
extra_axis_pos	external_axis_position	List[float]	—	External axis position

Legacy alias: CRIData = CriRealTimeData

## CriStatus

```
@dataclass
class CriStatus:
    status1_raw: int = 0
    status2_raw: int = 0
    project_running: bool = False
    project_stopped: bool = False
    project_paused: bool = False
    is_enabling: bool = False
```

```

is_disabled: bool = False
is_manual: bool = False
is_dragging: bool = False
is_moving: bool = False
collision_stop: bool = False
is_at_safe_pos: bool = False
has_alarm: bool = False
is_simulation: bool = False
is_emergency_stop: bool = False
is_rescue: bool = False
is_auto: bool = False
is_remote: bool = False
rt_control_mode: bool = False
error_code: int = 0

```

CRI status bit parsing.

Field	Description
project_running	Project running
project_stopped	Project stopped
project_paused	Project paused
is_enabling	Enabled
is_disabled	Disabled
is_manual	Manual mode
is_dragging	Drag mode
is_moving	Moving
collision_stop	Collision stopped
is_at_safe_pos	At safe position
has_alarm	Has alarm
is_simulation	Simulation mode
is_emergency_stop	Emergency stop pressed
is_rescue	Rescue mode
is_auto	Auto mode
is_remote	Remote mode
rt_control_mode	Real-time control mode
error_code	Error code (high 8 bits)

Legacy alias: `CRIStatus = CriStatus`

## Robot Settings

### RobotFrame

```
@dataclass
class RobotFrame:
    id: int
    x: float = 0.0
    y: float = 0.0
    z: float = 0.0
    a: float = 0.0
    b: float = 0.0
    c: float = 0.0
```

Tool / user coordinate single frame.

Field	Type	Description
id	int	Slot ID (0~15, 0 is reserved)
x, y, z	float	Position (mm)
a, b, c	float	Orientation (degrees)

```
frame = RobotFrame(id=1, x=100, y=0, z=50, a=0, b=0, c=0)
```

### RobotPayloadFrame

```
@dataclass
class RobotPayloadFrame:
    id: int
    m: float = 0.0
    mx: float = 0.0
    my: float = 0.0
    mz: float = 0.0
```

Payload coordinate single frame.

Field	Type	Description
id	int	Slot ID (0~15, 0 is reserved)
m	float	Mass (kg)
mx, my, mz	float	Center of mass (mm)

```
payload = RobotPayloadFrame(id=1, m=2.5, mx=0, my=0, mz=50)
```

## RobotParameters

```
@dataclass
class RobotParameters:
    default_tool_id: int = 0
    default_payload_id: int = 0
    default_coordinate_id: int = 0
    max_payload: float = 0.0
    tool: List[RobotFrame] = field(default_factory=list)
    payload: List[RobotPayloadFrame] = field(default_factory=list)
    coordinate: List[RobotFrame] = field(default_factory=list)
```

Settings interface parameter snapshot returned by `GetRobotParameters()`.

Field	Type	Description
default_tool_id	int	Default tool frame ID
default_payload_id	int	Default payload ID
default_coordinate_id	int	Default user coordinate ID
max_payload	float	Max payload (kg)
tool	List[RobotFrame]	Tool frame table (16 items)
payload	List[RobotPayloadFrame]	Payload frame table (16 items)
coordinate	List[RobotFrame]	User coordinate frame table (16 items)

## Global Variables

### GlobalVariable

```
@dataclass
class GlobalVariable:
    value: Any
    note: Optional[str] = None
```

Global variable data model.

Field	Type	Description
value	Any	Variable value (supports int, float, str, list, dict)
note	str	Variable note

```
var = GlobalVariable(value=42, note="counter")
robot.SaveGlobalVar("counter", var)
```

## Enum Types

### CoordinateType

```
class CoordinateType(str, Enum):
    USER = "user"
    TOOL = "tool"
```

Coordinate system type.

### IOType

```
class IOType(str, Enum):
    DI = "DI" # Digital Input
    DO = "DO" # Digital Output
    AI = "AI" # Analog Input
    AO = "AO" # Analog Output
```

### ExtendArrayType

```
class ExtendArrayType(str, Enum):
    BOOL = "Bool"
    UINT8 = "UInt8"
    INT8 = "Int8"
    UINT16 = "UInt16"
    INT16 = "Int16"
    UINT32 = "UInt32"
    INT32 = "Int32"
    FLOAT32 = "Float32"
```

Extended array data type.

### RS485BaudRate

```
class RS485BaudRate(IntEnum):
    B110 = 110
    B300 = 300
    B600 = 600
    B1200 = 1200
    B2400 = 2400
    B4800 = 4800
    B9600 = 9600
    B14400 = 14400
    B19200 = 19200
    B38400 = 38400
    B56000 = 56000
```

```
B57600 = 57600
B115200 = 115200
B128000 = 128000
B230400 = 230400
```

## RS485StopBits

```
class RS485StopBits(IntEnum):
    ONE = 1
    TWO = 2
```

## RS485Parity

```
class RS485Parity(IntEnum):
    NONE = 0
    ODD = 1
    EVEN = 2
```

## CriMask

```
class CriMask(IntFlag):
    TIMESTAMP = 1 << 0
    STATUS_1 = 1 << 1
    STATUS_2 = 1 << 2
    JOINT_POS = 1 << 8
    JOINT_VEL = 1 << 9
    CARTESIAN_POS = 1 << 10
    CARTESIAN_VEL = 1 << 11
    TCP_SPEED = 1 << 12
    JOINT_TORQUE = 1 << 13
    EXTERNAL_TORQUE = 1 << 14
    EXTRA_AXIS_POS = 1 << 15
```

CRI push bitmask.

Legacy alias: `CRIMask = CriMask`

## CriFilterType

```
class CriFilterType(IntEnum):
    NONE = 0
    AVERAGE = 1
    LOW_PASS = 2
    ELLIPTIC = 3
```

CRI real-time control filter type.

Legacy alias: `CRIFilterType = CriFilterType`

---

# Exceptions

## CodroidError

```
class CodroidError(Exception):  
    pass
```

Base exception. Parameter validation failure, illegal operations, etc.

## CodroidCommandException

```
class CodroidCommandException(CodroidError):  
    pass
```

Controller returned `err` field.

## CodroidNetworkError

```
class CodroidNetworkError(CodroidError):  
    pass
```

TCP connection or communication failure.

## CodroidTimeoutError

```
class CodroidTimeoutError(CodroidError):  
    pass
```

Operation timeout.



# CRI Real-time Data & Control API Reference

## CriRealtimePacketParser

```
class CriRealtimePacketParser:
    @staticmethod
    def parse(data: bytes) -> Optional[CriRealTimeData]
```

Parse 308-byte CRI UDP packets into `CriRealTimeData` (mm + degrees). Returns `None` for non-308-byte packets.

Automatic wire-layer unit conversion:

- Joint angles: rad → deg
- TCP position: m → mm
- TCP orientation: rad → deg
- Velocity: m/s → mm/s, rad/s → deg/s

```
from codroid import CriRealtimePacketParser

data = CriRealtimePacketParser.parse(raw_bytes)
if data is not None:
    print(f"Joint angles: {data.joint_position}")
    print(f"TCP pose: {data.tcp_pose}")
    print(f"Is moving: {data.status.is_moving}")
```

## CriStreamHandler

```
class CriStreamHandler:
    def __init__(
        self,
        high_precision: bool = True,
        mask: int = 0xFFFF,
        joint_count: int = 6,
        extra_axis_count: int = 0,
    )
```

Variable mask/precision CRI UDP packet parser. Supports flexible bitmask and precision configuration.

Parameter	Type	Default	Description
<code>high_precision</code>	<code>bool</code>	<code>True</code>	Double (Float64) / Single (Float32) precision
<code>mask</code>	<code>int</code>	<code>0xFFFF</code>	Bitmask controlling which fields to parse
<code>joint_count</code>	<code>int</code>	<code>6</code>	Number of joints
<code>extra_axis_count</code>	<code>int</code>	<code>0</code>	Number of extra axes

## bind

```
def bind(self, port: int) -> None
```

Bind UDP port.

## parse\_packet

```
def parse_packet(self, data: bytes) -> CriRealTimeData
```

Parse a single CRI data packet.

```
from codroid import CriStreamHandler

handler = CriStreamHandler(high_precision=True, mask=0xFFFF, joint_count=6)
handler.bind(10086)

try:
    while True:
        data, addr = handler._sock.recvfrom(2048)
        parsed = handler.parse_packet(data)
        print(f"Joint angles: {parsed.joint_position}")
finally:
    handler._sock.close()
```

---

## CriRealtimeDispatcher

```
class CriRealtimeDispatcher:
    def __init__(
        self,
        controller_ip: str,
        controller_udp_port: int = 9030,
        convert_to_si: bool = True,
    )
```

UDP real-time command dispatcher. Sends 64-byte control commands to the controller.

Parameter	Type	Default	Description
<code>controller_ip</code>	<code>str</code>	—	Controller IP
<code>controller_udp_port</code>	<code>int</code>	9030	Controller UDP port
<code>convert_to_si</code>	<code>bool</code>	True	Auto-convert mm/deg to m/rad

Supports `with` statement.

## SendCommand

```
def SendCommand(self, position6: Sequence[float], space: TrajectorySpace) -> None
```

Send a single real-time control frame.

Parameter	Type	Description
position6	Sequence[float]	6 position values (mm+deg or deg)
space	TrajectorySpace	JOINT or CARTESIAN

## SendTrajectory

```
def SendTrajectory(  
    self,  
    trajectory: Iterable[TrajectoryPoint],  
    space: TrajectorySpace,  
    period_ms: int,  
) -> None
```

Send a complete trajectory. Sends frame by frame at `period_ms` intervals.

```
from codroid import CriRealtimeDispatcher, TrajectorySpace  
  
with CriRealtimeDispatcher("192.168.1.136") as dispatcher:  
    dispatcher.SendCommand([0, 0, 90, 0, 90, 0], TrajectorySpace.JOINT)
```

---

## TrajectoryGenerator

```
class TrajectoryGenerator:  
    @staticmethod  
    def generate(  
        start: Sequence[float],  
        target: Sequence[float],  
        request: TrajectoryRequest,  
    ) -> List[TrajectoryPoint]  
  
    @staticmethod  
    def generate_multi_segment(  
        waypoints: Sequence[Sequence[float]],  
        request: TrajectoryRequest,  
    ) -> List[TrajectoryPoint]
```

Offline trajectory generation.

## generate

Single-segment trajectory generation. Both `start` and `target` are 6-element arrays.

## generate\_multi\_segment

Multi-segment trajectory generation. `waypoints` must contain at least 2 waypoints.

```
from codroid import (
    TrajectoryGenerator,
    TrajectoryRequest,
    TrajectorySpace,
    TrajectoryProfile,
)

request = TrajectoryRequest(
    space=TrajectorySpace.JOINT,
    frequency_hz=100,
    profile=TrajectoryProfile.CUBIC,
    acceleration=100,
    speed=40,
)

trajectory = TrajectoryGenerator.generate(
    [0, 0, 0, 0, 0, 0],
    [0, 0, 90, 0, 90, 0],
    request,
)

for point in trajectory:
    print(f"t={point.t:.3f}s, pos={point.position}")
```

## TrajectoryRequest

```
@dataclass
class TrajectoryRequest:
    space: TrajectorySpace
    frequency_hz: float
    profile: TrajectoryProfile
    acceleration: float
    speed: Optional[float] = None
    duration_seconds: Optional[float] = None
```

Trajectory generation request parameters.

Parameter	Type	Description
<code>space</code>	<code>TrajectorySpace</code>	<code>JOINT</code> (joint space) or <code>CARTESIAN</code> (Cartesian space)
<code>frequency_hz</code>	<code>float</code>	Sampling frequency (Hz)

Parameter	Type	Description
<code>profile</code>	<code>TrajectoryProfile</code>	<code>CUBIC</code> (cubic polynomial) or <code>TRAPEZOIDAL</code>
<code>acceleration</code>	<code>float</code>	Acceleration
<code>speed</code>	<code>float</code>	Speed (mutually exclusive with <code>duration_seconds</code> )
<code>duration_seconds</code>	<code>float</code>	Total duration (mutually exclusive with <code>speed</code> )

**Note:** Exactly one of `speed` and `duration_seconds` must be set.

## TrajectorySpace

```
class TrajectorySpace(Enum):
    JOINT = "Joint"
    CARTESIAN = "Cartesian"
```

## TrajectoryProfile

```
class TrajectoryProfile(Enum):
    CUBIC = "Cubic"
    TRAPEZOIDAL = "Trapezoidal"
```

## TrajectoryPoint

```
@dataclass
class TrajectoryPoint:
    t: float # Time (seconds)
    position: List[float] # 6 position values
```

## Complete CRI Control Flow Example

```
from codroid import (
    CodroidClient,
    CriRealtimeDispatcher,
    TrajectoryGenerator,
    TrajectoryRequest,
    TrajectorySpace,
    TrajectoryProfile,
    InitConsoleUtf8,
)

InitConsoleUtf8()
```

```

ROBOT_IP = "192.168.8.136"

with CodroidClient(host=ROBOT_IP) as robot:
    robot.ConnectRemoteAndSwitchOn()

    # 1. Start CRI real-time control
    robot.StartCriControl()

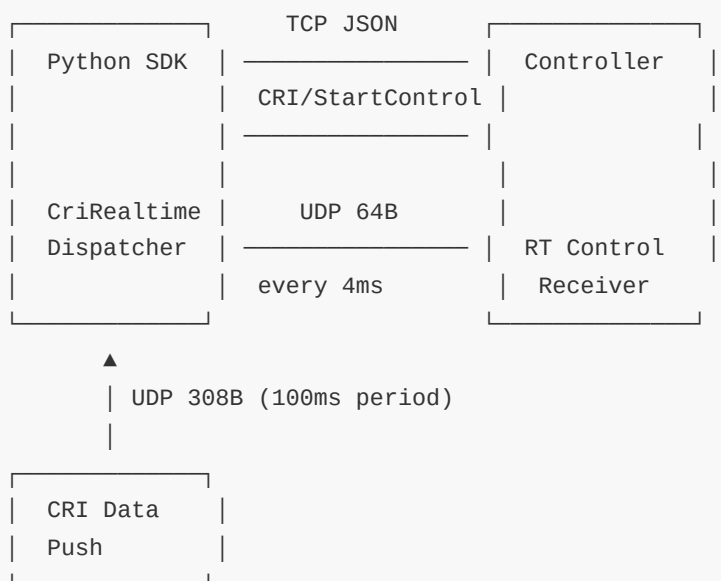
    # 2. Generate trajectory
    request = TrajectoryRequest(
        space=TrajectorySpace.JOINT,
        frequency_hz=250,
        profile=TrajectoryProfile.TRAPEZOIDAL,
        acceleration=100,
        speed=40,
    )
    trajectory = TrajectoryGenerator.generate(
        [0, 0, 0, 0, 0, 0],
        [0, 0, 90, 0, 90, 0],
        request,
    )

    # 3. Send trajectory
    with CriRealtimeDispatcher(ROBOT_IP) as dispatcher:
        dispatcher.SendTrajectory(trajectory, TrajectorySpace.JOINT, period_ms=4)

    # 4. Stop real-time control
    robot.StopCriControl()

```

## Workflow Diagram



# IO & Registers API Reference

---

## IO Operations

### Digital IO

#### GetDi

```
def GetDi(self, port: int) -> int
```

Get Digital Input (DI) value. Port 0~15, returns 0 or 1.

#### GetDo

```
def GetDo(self, port: int) -> int
```

Get Digital Output (DO) value. Port 0~15, returns 0 or 1.

#### SetDo

```
def SetDo(self, port: int, value: int) -> CommonResponse
```

Set Digital Output (DO) value. Port 0~15, value 0 or 1.

```
di0 = robot.GetDi(0)           # Read DI port 0
robot.SetDo(10, 1)             # Set DO port 10 to 1
robot.SetDo(10, di0)           # Write DI value to DO
```

---

## Analog IO

#### GetAi

```
def GetAi(self, port: int) -> float
```

Get Analog Input (AI) value. Port 0~3.

#### GetAo

```
def GetAo(self, port: int) -> float
```

Get Analog Output (AO) value. Port 0~3.

#### SetAo

```
def SetAo(self, port: int, value: float) -> CommonResponse
```

Set Analog Output (AO) value. Port 0~3.

```
ai0 = robot.GetAi(0)           # Read AI port 0
robot.SetAo(0, 3.14)           # Set AO port 0
```

## Batch IO

### GetIoValues

```
def GetIoValues(self, io_requests: List[Dict[str, Any]]) -> CommonResponse
```

Batch read multiple IO values.

Parameter	Type	Description
<code>io_requests</code>	<code>List[Dict]</code>	List containing <code>type</code> and <code>port</code>

```
response = robot.GetIoValues([
    {"type": "DI", "port": 0},
    {"type": "DI", "port": 1},
    {"type": "AI", "port": 0},
])
for item in response.db:
    print(f"{item['type']}{item['port']} = {item['value']}")
```

### SetIoValues

```
def SetIoValues(self, io_list: List[Dict[str, Any]]) -> List[CommonResponse]
```

Batch set IO values. Internally loops through `SetDo` / `SetAo`.

```
robot.SetIoValues([
    {"type": "DO", "port": 0, "value": 1},
    {"type": "DO", "port": 1, "value": 0},
    {"type": "AO", "port": 0, "value": 3.14},
])
```

## IO Port Ranges

Type	Port Range	Value Type
DI (Digital Input)	0~15	0 or 1
DO (Digital Output)	0~15	0 or 1
AI (Analog Input)	0~3	float
AO (Analog Output)	0~3	float

Out-of-range values throw `CodroidError`.



---

## Register Operations

### GetRegisterValue

```
def GetRegisterValue(self, address: int) -> Any
```

Get a single register value.

### GetRegisterValues

```
def GetRegisterValues(self, addresses: List[int]) -> CommonResponse
```

Batch get multiple register values.

```
val = robot.GetRegisterValue(0)
vals = robot.GetRegisterValues([0, 1, 2])
for item in vals.db:
    print(f"R{item['address']} = {item['value']}")
```

### SetRegisterValue

```
def SetRegisterValue(self, address: int, value: Any) -> CommonResponse
```

Write register value.

```
robot.SetRegisterValue(0, 42)
robot.SetRegisterValue(1, 3.14)
```

---

## Extended Arrays

### SetExtendArrayType

```
def SetExtendArrayType(self, index: int, data_type: ExtendArrayType) -> CommonResponse
```

Set extended array data type. `index` range 0~999.

Data Type	Description
<code>ExtendArrayType.BOOL</code>	Boolean
<code>ExtendArrayType.UINT8</code>	Unsigned 8-bit integer
<code>ExtendArrayType.INT8</code>	Signed 8-bit integer
<code>ExtendArrayType.UINT16</code>	Unsigned 16-bit integer
<code>ExtendArrayType.INT16</code>	Signed 16-bit integer

Data Type	Description
<code>ExtendArrayType.UINT32</code>	Unsigned 32-bit integer
<code>ExtendArrayType.INT32</code>	Signed 32-bit integer
<code>ExtendArrayType.FLOAT32</code>	32-bit float

```
from codroid import ExtendArrayType

robot.SetExtendArrayType(0, ExtendArrayType.FLOAT32)
```

### RemoveExtendArray

```
def RemoveExtendArray(self, index: int) -> CommonResponse
```

Remove extended array index (reset data). `index` range 0~999.

## Complete IO + Register Example

```
from codroid import CodroidClient, ExtendArrayType, InitConsoleUtf8

InitConsoleUtf8()

with CodroidClient(host="192.168.8.136") as robot:
    robot.ConnectRemoteAndSwitchOn()

    # --- IO ---
    # Read single DI
    di0 = robot.GetDi(0)
    print(f"DI 0 = {di0}")

    # Write single DO
    robot.SetDo(10, 1)

    # Batch read
    io_data = robot.GetIoValues([
        {"type": "DI", "port": 0},
        {"type": "DI", "port": 1},
        {"type": "AI", "port": 0},
    ])
    for item in io_data.db:
        print(f" {item['type']}{item['port']} = {item['value']}")

    # --- Registers ---
    # Read single
    val = robot.GetRegisterValue(0)
    print(f"R0 = {val}")

    # Write
```

```
robot.SetRegisterValue(0, val + 1)

# Batch read
regs = robot.GetRegisterValues([0, 1, 2])
for item in regs.db:
    print(f" R{item['address']} = {item['value']}")

# Extended arrays
robot.SetExtendArrayType(0, ExtendArrayType.FLOAT32)
robot.RemoveExtendArray(0)
```

# Utilities API Reference

---

## Publish / Subscribe

### PublishTopics

```
class PublishTopics:
    PROJECT_STATE = "publish/ProjectState"
    VAR_UPDATE = "publish/VarUpdate"
    ROBOT_STATUS = "publish/RobotStatus"
    ROBOT_POSTURE = "publish/RobotPosture"
    ROBOT_COORDINATE = "publish/RobotCoordinate"
    LOG = "publish/Log"
    ERROR = "publish/Error"
```

Controller push topic constants.

---

### PublishNotification

```
@dataclass
class PublishNotification:
    ty: str          # Topic type
    db: Any          # Push data
    raw_json: str    # Raw JSON string
```

### PublishTopicSubscription

```
class PublishTopicSubscription:
    def dispose(self) -> None
    def Dispose(self) -> None # C# alias
```

Subscription handle. Call `dispose()` to remove local callback (does not send unsubscribe to controller).

---

### SubscribePublishTopic (CodroidClient only)

```
def SubscribePublishTopic(
    self,
    topic_ty: str,
    handler: Callable[[PublishNotification], None],
    tc_milliseconds: int = 100,
) -> PublishTopicSubscription
```

Subscribe to controller push topic. On first local subscription, automatically sends subscription request to controller.

```
from codroid import CodroidClient, PublishTopics, InitConsoleUtf8
```

```

InitConsoleUtf8()

def on_status(notification):
    print(f"Received {notification.ty}: {notification.db}")

def on_error(notification):
    print(f"Error: {notification.db}")

with CodroidClient(host="192.168.8.136") as robot:
    robot.ConnectRemoteAndSwitchOn()

    sub1 = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_status)
    sub2 = robot.SubscribePublishTopic(PublishTopics.ERROR, on_error)

    # ... run ...

    sub1.dispose()
    sub2.dispose()

```

## Global Variables

### GetGlobalVars

```
def GetGlobalVars(self) -> CommonResponse
```

Get all global variables.

### GetGlobalVarsCatalog

```
def GetGlobalVarsCatalog(self) -> CommonResponse
```

Get global variables catalog (same TCP request as `GetGlobalVars`).

### SaveGlobalVar

```
def SaveGlobalVar(self, name: str, variable: GlobalVariable) -> CommonResponse
```

Save a single global variable.

### SaveGlobalVars

```
def SaveGlobalVars(self, variables: Dict[str, GlobalVariable]) -> CommonResponse
```

Batch save global variables. Variable names are validated against Lua reserved words.

## RemoveGlobalVars

```
def RemoveGlobalVars(self, names: List[str]) -> CommonResponse
```

Remove global variables.

## GetProjectVar

```
def GetProjectVar(self) -> CommonResponse
```

Get all current project variable values (only valid when project is running).

```
from codroid import GlobalVariable

# Save
robot.SaveGlobalVar("counter", GlobalVariable(value=0, note="counter"))
robot.SaveGlobalVar("name", GlobalVariable(value="test"))

# Batch save
robot.SaveGlobalVars({
    "x": GlobalVariable(value=100.0),
    "y": GlobalVariable(value=200.0),
})

# Read
response = robot.GetGlobalVars()
print(response.db)

# Remove
robot.RemoveGlobalVars(["counter", "name"])

# Get project variables (while project is running)
proj_vars = robot.GetProjectVar()
```

---

## Kinematics

### AposToCpos

```
def AposToCpos(
    self,
    jp: Sequence[float],
    coor: Optional[Sequence[float]] = None,
    tool: Optional[Sequence[float]] = None,
    ep: Sequence[float] = [],
) -> CommonResponse
```

Forward kinematics (joint → Cartesian). `jp` is 6 joint angles (degrees).

Parameter	Type	Description
<code>jp</code>	<code>Sequence[float]</code>	6 joint angles (degrees)
<code>coor</code>	<code>Sequence[float]</code>	User coordinate <code>[x, y, z, a, b, c]</code>
<code>tool</code>	<code>Sequence[float]</code>	Tool coordinate <code>[x, y, z, a, b, c]</code>
<code>ep</code>	<code>Sequence[float]</code>	External axis positions

```
response = robot.AposToCpos([0, 0, 90, 0, 90, 0])
print(f"TCP pose: {response.db}")
```

## CposToApos

```
def CposToApos(
    self,
    cp: Sequence[float],
    rj: Optional[Sequence[float]] = None,
    ep: Sequence[float] = [],
) -> CommonResponse
```

Inverse kinematics (Cartesian → joint). `cp` is `[x, y, z, a, b, c]` (mm + degrees). `rj` is reference joint angles (default `[20, 20, 20, 20, 20, 20]`).

```
response = robot.CposToApos([400, 200, 500, 180, 0, 90])
print(f"Joint angles: {response.db}")
```

## CalculateRelativePose

```
def CalculateRelativePose(
    self,
    pos: Sequence[float],
    offset: Sequence[float],
    coor_type: CoordinateType = CoordinateType.TOOL,
    pos_coor: Optional[Sequence[float]] = None,
    coor: Optional[Sequence[float]] = None,
) -> CommonResponse
```

Cartesian coordinate offset calculation.

Parameter	Type	Description
<code>pos</code>	<code>Sequence[float]</code>	Current TCP pose
<code>offset</code>	<code>Sequence[float]</code>	Offset values
<code>coor_type</code>	<code>CoordinateType</code>	<code>USER</code> or <code>TOOL</code>
<code>pos_coor</code>	<code>Sequence[float]</code>	Current TCP coordinate

Parameter	Type	Description
<code>coord</code>	<code>Sequence[float]</code>	Offset coordinate

```
from codroid import CoordinateType

response = robot.CalculateRelativePose(
    pos=[400, 200, 500, 180, 0, 90],
    offset=[0, 0, -100, 0, 0, 0],
    coord_type=CoordinateType.TOOL,
)
print(f"Offset pose: {response.db}")
```

## CposToCpos / CposToCposPose (2.1.10+)

Coordinate system transformation: convert a TCP pose from coordinate system 1 + tool 1 to coordinate system 2 + tool 2. Protocol `Robot/cpostocpos`.

```
def CposToCpos(
    self,
    cp: Sequence[float],
    coord1: Sequence[float],
    tool1: Sequence[float],
    coord2: Sequence[float],
    tool2: Sequence[float],
) -> List[float]
```

Parameter	Type	Description
<code>cp</code>	<code>Sequence[float]</code>	Current TCP pose <code>[x, y, z, a, b, c]</code> (mm+deg)
<code>coord1</code>	<code>Sequence[float]</code>	Source coordinate system <code>[x, y, z, a, b, c]</code>
<code>tool1</code>	<code>Sequence[float]</code>	Source tool <code>[x, y, z, a, b, c]</code>
<code>coord2</code>	<code>Sequence[float]</code>	Target coordinate system <code>[x, y, z, a, b, c]</code>
<code>tool2</code>	<code>Sequence[float]</code>	Target tool <code>[x, y, z, a, b, c]</code>

```
result = robot.CposToCpos(
    cp=[400, 200, 500, 180, 0, 90],
    coord1=[0,0,0,0,0,0], tool1=[0,0,0,0,0,0],
    coord2=[100,0,0,0,0,0], tool2=[0,0,100,0,0,0]
)
print(f"Converted pose: {result}")
```

`CposToCposPose` returns `CartesianPoint`:



```
pose = robot.CposToCposPose(
    cp=CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]),
    coor1=[0,0,0,0,0,0], tool1=[0,0,0,0,0,0],
    coor2=[100,0,0,0,0,0], tool2=[0,0,100,0,0,0]
)
```

## ConsoleUtf8

### InitConsoleUtf8

```
def InitConsoleUtf8() -> None
```

Windows console UTF-8 initialization. Must be called at entry when running examples with Chinese characters in `cmd` (not Windows Terminal). No-op on Linux / macOS.

```
from codroid import InitConsoleUtf8

InitConsoleUtf8()
```

Legacy alias: `init_console_utf8 = InitConsoleUtf8`

## PrintBanner

```
from codroid import PrintBanner

PrintBanner("Title", "Subtitle")
```

Print a colored banner title in the terminal. Requires `colorama`: `pip install "codroid-robot-sdk[color]"`.

## Complete Utilities Example

```
from codroid import (
    CodroidClient,
    PublishTopics,
    GlobalVariable,
    CoordinateType,
    InitConsoleUtf8,
    PrintBanner,
)

InitConsoleUtf8()
PrintBanner("Utilities Example", "Publish / Global Vars / Kinematics")

ROBOT_IP = "192.168.8.136"

# --- Publish / Subscribe ---
```

```

def on_status(notification):
    print(f"[Publish] {notification.ty}: {notification.db}")

with CodroidClient(host=ROBOT_IP) as robot:
    robot.ConnectRemoteAndSwitchOn()

    sub = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_status)

    # --- Global Variables ---
    robot.SaveGlobalVar("test_var", GlobalVariable(value=42, note="test"))
    response = robot.GetGlobalVars()
    print(f"Global vars: {response.db}")
    robot.RemoveGlobalVars(["test_var"])

    # --- Kinematics ---
    fk = robot.AposToCpos([0, 0, 90, 0, 90, 0])
    print(f"FK result: {fk.db}")

    ik = robot.CposToApos([400, 200, 500, 180, 0, 90])
    print(f"IK result: {ik.db}")

    rel = robot.CalculateRelativePose(
        pos=[400, 200, 500, 180, 0, 90],
        offset=[0, 0, -100, 0, 0, 0],
        coord_type=CoordinateType.T00L,
    )
    print(f"Offset result: {rel.db}")

    sub.dispose()

```