

Joint and Kinematics

1. Basic Motion Routine

- **Parallel Joint Space (PJS)**
 - **Parallel Joint Space (PJS)** is a control mode that operates each robot joint using the functional coordinate system defined by the **Pitch (P)** and **Roll (R)** joints. Because PJS uses the same P/R axes defined in the robot's URDF model, users can issue commands using a conventional joint-based control approach without additional coordinate conversion.
- **Motor Space (MS)**
 - **Motor Space (MS)** is a low-level control mode that directly controls the physical **Motor A** and **Motor B** actuators inside the joint. Due to the characteristics of the parallel mechanism, functional **Pitch–Roll (PR)** motion is generated through the combined operation of Motors A and B. Therefore, when using MS mode, users must perform the **PR ↔ AB conversion** themselves using forward and inverse kinematics.

2. Joint Motor Sequence

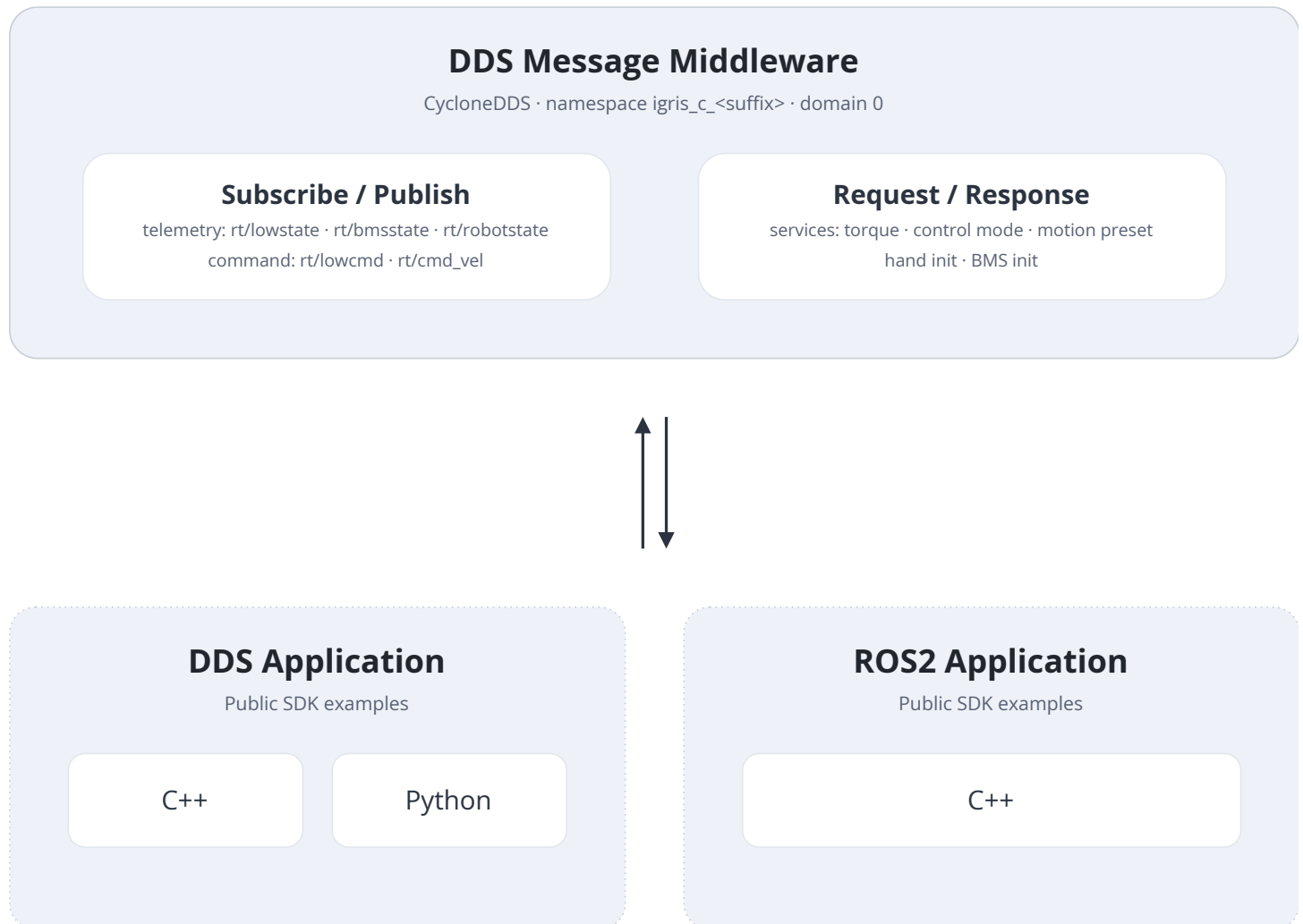
- **IGRIS-C Joint Motor Order**

Motor ID	Motor Name	Joint Name
1	Waist_Yaw	Waist_Yaw
2	Waist_L	Waist_Roll
3	Waist_R	Waist_Pitch
4	Hip_Pitch_L	Hip_Pitch_L
5	Hip_Roll_L	Hip_Roll_L
6	Hip_Yaw_L	Hip_Yaw_L
7	Knee_Pitch_L	Knee_Pitch_L
8	Ankle_Out_L	Ankle_Pitch_L
9	Ankle_In_L	Ankle_Roll_L
10	Hip_Pitch_R	Hip_Pitch_R
11	Hip_Roll_R	Hip_Roll_R
12	Hip_Yaw_R	Hip_Yaw_R
13	Knee_Pitch_R	Knee_Pitch_R
14	Ankle_Out_R	Ankle_Pitch_R
15	Ankle_In_R	Ankle_Roll_R
16	Shoulder_Pitch_L	Shoulder_Pitch_L
17	Shoulder_Roll_L	Shoulder_Roll_L
18	Shoulder_Yaw_L	Shoulder_Yaw_L
19	Elbow_Pitch_L	Elbow_Pitch_L
20	Wrist_Yaw_L	Wrist_Yaw_L
21	Wrist_Front_L	Wrist_Roll_L
22	Wrist_Back_L	Wrist_Pitch_L
23	Shoulder_Pitch_R	Shoulder_Pitch_R
24	Shoulder_Roll_R	Shoulder_Roll_R

25	Shoulder_Yaw_R	Shoulder_Yaw_R
26	Elbow_Pitch_R	Elbow_Pitch_R
27	Wrist_Yaw_R	Wrist_Yaw_R
28	Wrist_Front_R	Wrist_Roll_R
29	Wrist_Back_R	Wrist_Pitch_R
30	Neck_Yaw	Neck_Yaw
31	Neck_Pitch	Neck_Pitch

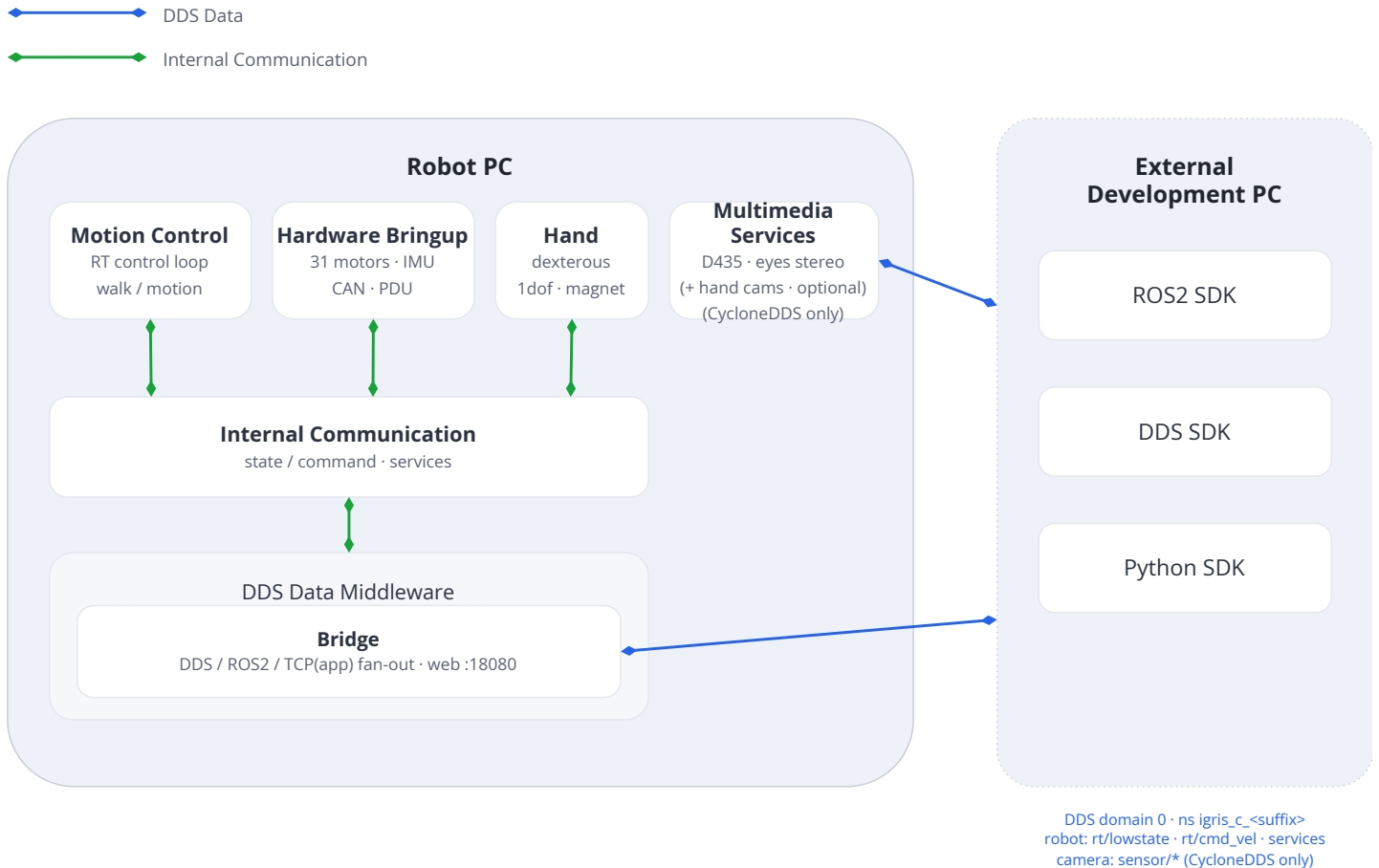
SDK Overview

1. Middleware



- IGRIS-C uses DDS as its message middleware, employing two main data communication methods: Publish/Subscribe mode and Request/Response mode.
- **Publish/Subscription**
 - Receivers subscribe to messages, and senders transmit messages to receivers according to the subscription list. This method is primarily used for **medium-to-high frequency or continuous data exchange**.
- **Request/Response**
 - Data queries or actions are performed through requests. This method is used for **low-frequency data exchange or function switching**.

2. Architecture Description



- Commands and state information delivered through the SDK are primarily implemented via DDS. The DDS-based communication structure enables reliable, real-time data exchange between modules.
- **DDS Middleware**
 - IGRIS-C and its related SDKs exchange commands and state information on top of this DDS middleware, providing a scalable and consistent communication structure between internal modules and external development PCs.
- **Motion Control** (the low-level control engine that directly generates and maintains the robot's motion)
 - The Motion Control module is IGRIS-C's **core low-level motion control engine**, operating based on a very fast internal control loop.
- **Key Features**
 1. **High-frequency low-level control loop**
 - Control loop execution at μ s–ms intervals in the internal control code
 - High-speed updates of torque, velocity, and position commands for each joint/motor
 2. **CAN communication-based motor control**
 - Encodes received control commands into CAN messages and delivers them to each motor

- Collects motor status (current, velocity, temperature, angle, etc.) in real time and transmits it to DDS

3. State information → delivered to DDS middleware

- Various sensor values such as IMU, joint angles, and dynamic states are published as DDS topics and can be subscribed to from the development PC

4. Processing commands received from DDS

- Receives DDS motion commands published by PC2 or an external development PC
- Applies them immediately within the control loop and reflects them to the motors

- **Image Services**

- The Image Services module handles **image stream processing and external communication** for IGRIS-C.
- Key Features

1. Camera image processing and streaming

- Collects image frames from cameras connected to IGRIS-C
- Transmits directly via DDS

***“Supported interfaces:** The SDK provides three client interfaces — **DDS (Cyclone DDS)**, **ROS2**, and **Python**. Examples are located in the [examples/cyclonedds/](#), [examples/ros2/](#), and [examples/python/](#) folders respectively. (The previous documentation's notice of "ROS2 planned for later" is no longer valid — ROS2 is now officially supported.)”*

3. Robot Namespace (topic prefix)

All DDS topics are resolved under a namespace specific to each robot **unit**. Each unit has a different namespace, in the form of the `igris_c_` prefix followed by a **per-robot suffix** (which varies by unit). Example: `igris_c_IG01`.

- In the SDK, the namespace is specified using `ChannelFactory::Init(domain_id, namespace)`.
- In code, topics are written as `rt/...`, but are actually resolved as `<namespace>/rt/...`.
 - Example: with namespace `igris_c_IG01` → `rt/lowstate` resolves to `igris_c_IG01/rt/lowstate`

```
// C++
ChannelFactory::Instance()->Init(0, "igris_c_IG01");           // domain 0, unit IG01
Subscriber<LowState> sub("rt/lowstate", QosProfile::SensorData()); //
igris_c_IG01/rt/lowstate
```

C++

```
# Python
channel = igc_sdk.ChannelFactory.Instance()
channel.Init(0, "igris_c_IG01")
```

“▲ The development PC (SDK client) and the robot must use the **same namespace + domain_id** for communication to work. Verify and set the namespace of the unit you intend to connect to.”

Before running the examples — the `CYCLONEDDS_URI` environment variable

DDS communication uses a **CycloneDDS XML** file containing the network interface configuration. When running SDK examples on an external development PC, this XML must be specified via `CYCLONEDDS_URI` to connect to the robot's DDS network. An example `cyclonedds.xml` is bundled **at the root** of the SDK distribution.

1. Edit the `<NetworkInterface name="...">` in `cyclonedds.xml` to the name of the **network interface on this PC that communicates with the robot**. (To check available interfaces: `ip link`, or set `autodetermine="true"` for automatic selection.)
2. Export `CYCLONEDDS_URI` to point to this file before running the examples.

C++

```
# From the root of the SDK distribution (using the bundled cyclonedds.xml)
export CYCLONEDDS_URI=file://$(pwd)/cyclonedds.xml
export ROS_DOMAIN_ID=0      # When using ROS2 examples (domain must match)

# Then run the example
./cyclonedds_hand_example 0 igris_c_IG01
```

“If this is not set, or the interface is incorrect, participant discovery will fail and topics will not be visible. Namespace and domain_id are specified via example arguments (or `ChannelFactory::Init`), not the XML file.”

4. Get SDK

- **SDK Introduction**

- `igris_c_sdk` is a next-generation robot SDK developed by ROBROS. This SDK encapsulates interfaces such as basic motor control and advanced motion control, and provides related functional interfaces.
- By referring to the provided SDK and examples, you can learn robot control and complete Secondary Development for IGRIS-C.

- **SDK Download**

- SDK: GitHub - robrosinc/igris_c_sdk_public: IGRIS_C SDK
- Description: GitHub - robrosinc/igris_c_description_public

- **URDF**

- URDF files for all IGRIS-C versions can be downloaded from the Description repository above.

ControlMode

*"This page reorganizes the former "Low Level Control" page into **Control Mode**, summarizing the meaning and switching methods of each control mode supported by IGRIS-C, as well as the message interfaces required for low-level control."*

- IGRIS-C's behavior hierarchy is switched via **Control Mode**. Mode-switch commands are sent to the controller through the SDK's `IgrisC_Client::SendControlModeCommand()` (internally a `ControlModeCommandRequest` Request/Response), and the controller maps this to an internal `RobotControlMode` to change the control loop's behavior.

```
struct ControlModeCommandRequest {  
    Header          header;  
    string          request_id; // ID for matching requests  
    ControlModeCommandType command_type; // Mode to switch to (see list below)  
    string          preset_id; // MOTION_PRESET only  
    boolean         is_cyclic;  // MOTION_PRESET_CYCLIC_TOGGLE only  
};  
// Response: ServiceResponse { request_id, success, message, error_code(0=success) }
```

C++

1. List of Control Modes

`ControlModeCommandType` (19 types total, values 0–18). Controller behavior is based on `robotHandler` dispatch.

#	command_type	Controller mapping/behavior	Layer
0	MOTION_PRESET	startMotionPreset(preset_id)	High-Level
1	MOTION_PRESET_CYCLIC_TOGGLE	setMotionPresetCyclic(is_cyclic)	High-Level
2	JOINT_POSITION_HOLD	→ JOINT_HOLD_MODE	-
3	MOTION_STOP	holdCurrentPosition()	High-Level
4	WALKMODE_ON	→ WALK_MODE	-
5	LOW_LEVEL_JOINT_CONTROL	→ LOW_LEVEL_MODE	Low-Level
6	LOW_LEVEL_WALKMODE_ON	→ LOW_LEVEL_WALK_MODE	Low-Level
7	HIGH_LEVEL_WALKMODE_ON	→ HIGH_LEVEL_WALK_MODE	High-Level
8	HIGH_LEVEL_JOINT_CONTROL	→ HIGH_LEVEL_MODE	High-Level
9–18	CUSTOM_MODE_1 – CUSTOM_MODE_10	(internal use only)	Not provided

"command_type values follow the format

`igris_c::msg::dds::ControlModeCommandType::CONTROL_MODE_CMD_<NAME>` (e.g., `CONTROL_MODE_CMD_MOTION_PRESET`)."

“▲ `CUSTOM_MODE_1` – `CUSTOM_MODE_10` (9–18) are **not currently provided to users**. These are internal/extension-only entries with reserved slots in the enum only — do not use them when working with the SDK.”

2. Description of Each Mode

Low-Level Control

The SDK **directly** commands each motor's targets ($q \cdot dq \cdot \tau \cdot kp \cdot kd$) via `rt/lowcmd` . In this mode, control loop responsibility resides externally (with the SDK). (See c.5 for the message interface.)

- `LOW_LEVEL_JOINT_CONTROL` (5) — `LOW_LEVEL_MODE` . Direct joint control via SDK LowCmd.
- `LOW_LEVEL_WALKMODE_ON` (6) — `LOW_LEVEL_WALK_MODE` . Walking mode based on low-level control.

High-Level Control

The controller's built-in logic receives higher-level commands and generates/maintains motion. **Motion preset execution is only possible at this layer.**

- `HIGH_LEVEL_JOINT_CONTROL` (8) — `HIGH_LEVEL_MODE` . Controller's high-level joint control.
- `HIGH_LEVEL_WALKMODE_ON` (7) — `HIGH_LEVEL_WALK_MODE` . High-level walking mode.

Walk

- `WALKMODE_ON` (4) — `WALK_MODE` . Enters walking mode.

Motion Preset

- `MOTION_PRESET` (0) — Plays back a predefined motion specified by `preset_id` (`startMotionPreset`). Can only be started from `HIGH_LEVEL_MODE` or `HIGH_LEVEL_WALK_MODE` (rejected in other modes). Populate the `preset_id` argument in the request.
- `MOTION_PRESET_CYCLIC_TOGGLE` (1) — Toggles whether the preset repeats cyclically (`setMotionPresetCyclic`). Uses the `is_cyclic` argument in the request.

Stop / Hold

- `JOINT_POSITION_HOLD` (2) — `JOINT_HOLD_MODE` . Maintains (holds) the current joint positions.
- `MOTION_STOP` (3) — Stops the ongoing motion and maintains the current posture (`holdCurrentPosition`). Only supported from `HIGH_LEVEL_MODE` / `HIGH_LEVEL_WALK_MODE` .

Custom — Currently Not Provided

- `CUSTOM_MODE_1` – `CUSTOM_MODE_10` (9–18) — **Internal/extension-only slots not currently provided to users.** These are defined in the enum but are not modes callable by SDK users, so do not use them in normal control.

3. Mode Switching API (SDK)

Mode switching is performed via the single method `IgrisC_Client::SendControlModeCommand()` .

C++

```
ServiceResponse SendControlModeCommand(
    ControlModeCommandType command_type,
    const std::string& preset_id = "", // MOTION_PRESET only
    bool is_cyclic = false, // MOTION_PRESET_CYCLIC_TOGGLE only
    int timeout_ms = 5000);
```

Usage example (based on [examples/cyclonedds/cyclonedds_service.cpp](#)):

C++

```
using igris_c::msg::dds::ControlModeCommandType;

// Execute motion preset "HOME" (at the HIGH_LEVEL layer)
res = client.SendControlModeCommand(
    ControlModeCommandType::CONTROL_MODE_CMD_MOTION_PRESET, "HOME", false,
    60000);

// Hold current position
res = client.SendControlModeCommand(
    ControlModeCommandType::CONTROL_MODE_CMD_JOINT_POSITION_HOLD, "", false,
    60000);

// Stop ongoing motion
res = client.SendControlModeCommand(
    ControlModeCommandType::CONTROL_MODE_CMD_MOTION_STOP, "", false, 60000);

// Switch to low-level joint control mode (use rt/lowcmd afterward)
res = client.SendControlModeCommand(
    ControlModeCommandType::CONTROL_MODE_CMD_LOW_LEVEL_JOINT_CONTROL, "",
    false, 60000);
```

Check the result via the returned `ServiceResponse`'s `success` / `error_code` (0=success) / `message`.

4. Service Topics Summary

Request/Response service channels used by the SDK client:

Function	Service Topic	SDK Method
BMS/motor initialization	<code>rt/service/bms_init</code>	<code>InitBms()</code>
Torque ON/OFF	<code>rt/service/torque</code>	<code>SetTorque()</code>
Hand initialization	<code>rt/service/hand_init</code>	<code>InitHand()</code>
Control mode switching	<code>rt/service/control_mode</code>	<code>SendControlModeCommand()</code>
Mujoco simulation control (sim only)	<code>rt/service/mujoco_sim</code>	<code>SendMujocoSimCmd()</code>

“Each service operates as a pair: `<topic>/request` (publishes the request) · `<topic>/response` (subscribes to the response). Example: `rt/service/bms_init/request` · `rt/service/bms_init/response` . (The table above shows the base topic.)”

Low-level streams: `rt/lowcmd` (LowCmd, commands) / `rt/lowstate` (LowState, status).

5. Low-Level Control Message Interface

IGRIS-C Topic List

- Low-level Cmd & State (high frequency): `rt/lowcmd` / `rt/lowstate`
- Service State (low frequency)

Common (`igris_c::msg::dds`)

```
const uint32 N_JOINTS = 31; // Fixed DOF Num
```

```
// Kinematic Space (MS=Motor Space/AB, PJS=Parallel Joint Space/PR)
enum KinematicMode { MS, PJS };
```

```
// Motor control is HYBRID mode only. (The former MotorMode { CURRENT, HYBRID }
enum has been removed.)
```

C++

Command Side

```
struct MotorCmd {          // HYBRID only
    uint16 id; // MS: motor index / PJS: virtual joint index
    float q; // Position (rad)
    float dq; // Velocity (rad/s)
    float tau; // Feedforward Torque (Nm)
    float kp; // kP
    float kd; // kD
};

struct LowCmd {
    Header header;
    KinematicMode kinematic_modes[5]; // Per parallel-link group: waist / L_ankle / R_ankle /
    L_wrist / R_wrist
    MotorCmd motors[31]; // Fixed length, 31 DOF
};
```

```
// Usage example: set all groups to the same mode
cmd.kinematic_modes().fill(KinematicMode::PJS);
```

State Side

C++

```
struct IMUState {
    float quaternion[4];    // w, x, y, z
    float gyroscope[3];     // rad/s
    float accelerometer[3]; // m/s^2
    float rpy[3];           // rad (intrinsic ZYX)
};

struct MotorState {
    float q;                // Position (rad)
    float dq;               // Velocity (rad/s)
    float tau_est;          // Estimated Torque (Nm)
    int16 temperature;      // Motor Temperature (°C)
    uint32 status_bits;     // Status Bits (fault/limit, see separate document)
};

struct JointState {
    float q;                // PJS Position (rad)
    float dq;               // PJS Velocity (rad/s)
    float tau_est;          // PJS Estimated Torque (Nm)
    uint32 status_bits;     // Status Bits
};

struct LowState {
    Header header;
    IMUState imu_state;      // Base IMU state
    MotorState motor_state[N_JOINTS]; // MS raw state
    JointState joint_state[N_JOINTS]; // PJS derived state
};
```

"All types are defined in the `igris_c::msg::dds` namespace (the IDL's `module`). In actual use, reference them as `igris_c::msg::dds::TypeName`."

6. System & Mode Control API (Initialization / Torque / Hand)

C++

```
#include <igris_c_sdk/channel_factory.hpp>
#include <igris_c_sdk/igris_c_client.hpp>
using namespace igris_c_sdk;
using namespace igris_c::msg::dds;

int main() {
    int domain_id = 0;
    ChannelFactory::Instance()->Init(domain_id, "igris_c_IG01"); // domain + robot unit
    namespace (see b.3)
    if (!ChannelFactory::Instance()->IsInitialized()) { return 1; }

    Igrisc_Client client;
    client.Init();
    client.SetTimeout(10.0f); // 10-second timeout
    // ... subsequent service calls
}
```

BMS / Motor Initialization

C++

```
ServiceResponse res = client.InitBms(BmsInitType::BMS_INIT, 30000); // BMS power
ON
res = client.InitBms(BmsInitType::MOTOR_INIT, 30000); // Motor initialization
only
res = client.InitBms(BmsInitType::BMS_AND_MOTOR_INIT, 30000); // Power ON +
motor (full sequence)
res = client.InitBms(BmsInitType::BMS_INIT_NONE, 30000); // BMS power OFF
```

- `init_type` (`BmsInitType`) — Values actually handled by the bridge:
 - `BMS_INIT_NONE` (0) — **BMS power OFF** (`setBmsPowerAsync(false)`)
 - `BMS_INIT` (1) — BMS power ON
 - `MOTOR_INIT` (2) — Motor initialization
 - `BMS_AND_MOTOR_INIT` (3) — Power ON + motor initialization (full sequence)
 - `BMS_OFF` (4) — Defined in the enum but **currently unhandled by the bridge (error_code -2)** → do not use

- `timeout_ms` — Timeout in ms (default 5000)
- Returned `ServiceResponse` — `request_id` , `success` , `message` , `error_code` (0=success)

Torque Control

```
ServiceResponse res = client.SetTorque(TorqueType::TORQUE_ON, 30000);
res = client.SetTorque(TorqueType::TORQUE_OFF, 30000);
```

C++

- `torque` (`TorqueType`) — `TORQUE_NONE` (0, reserved) / `TORQUE_ON` (1) / `TORQUE_OFF` (2)
- `timeout_ms` — Timeout in ms (default 5000)

Hand Initialization

```
ServiceResponse res = client.InitHand(30000); // End-effector calibration/homing
```

C++

(For detailed hand control, see the [e. Hand](#) page)

BMS Status Query (`rt/bmsstate` , low frequency)

`tick` , `body_power` (RELAY_OFF/ON), `legs_power` , `estop` , `connect` , `battery` (voltage),
`bms_init_state`

7. Python Binding

Installing the `dist/*.whl` from the `igris_c_sdk_public` repository into a Python environment allows usage with the same getter/setter pattern as C++. Examples are located in the [examples/python](#) folder.

```
import igris_c_sdk as igc_sdk

channel_instance = igc_sdk.ChannelFactory.Instance()
channel_instance.Init(0, "igris_c_IG01") # domain_id + robot unit namespace (see b.3)

client = igc_sdk.IgrisC_Client()
client.Init()
client.SetTimeout(20.0)
```

C++

Real-time lowstate Subscription

C++

```
lowstate_subscriber = igc_sdk.LowStateSubscriber("rt/lowstate",
igc_sdk.QosProfile.SensorData())

def lowstate_callback(msg):
    imu = msg.imu_state()
    q = imu.quaternion(); gyro = imu.gyroscope(); acc = imu.accelerometer(); rpy =
imu.rpy()
    m0 = msg.motor_state()[0]
    print(m0.q(), m0.dq(), m0.tau_est(), m0.temperature(), m0.status_bits())

lowstate_subscriber.init(lowstate_callback)
```

Service Calls (init / torque / control mode)

C++

```
# BMS / Motor
client.InitBms(igc_sdk.BmsInitType.BMS_INIT, 5000)
client.InitBms(igc_sdk.BmsInitType.BMS_AND_MOTOR_INIT, 5000)

# Torque
client.SetTorque(igc_sdk.TorqueType.TORQUE_ON, 5000)

# Control mode (same as C++ SendControlModeCommand)
client.SendControlModeCommand(
    igc_sdk.ControlModeCommandType.CONTROL_MODE_CMD_MOTION_PRESET, "HOME",
    False, 60000)
client.SendControlModeCommand(
    igc_sdk.ControlModeCommandType.CONTROL_MODE_CMD_JOINT_POSITION_HOLD, "",
    False, 60000)

# Hand
client.InitHand(5000)
```

Sending LowCmd (e.g., controlling neck yaw/pitch motors)

C++

```
lowcmd_publisher = igc_sdk.LowCmdPublisher("rt/lowcmd",
igc_sdk.QosProfile.SensorData())
lowcmd_publisher.init()

low_cmd_msg = igc_sdk.LowCmd()
for i in range(0, 31):
    motor_cmd = low_cmd_msg.motors()[i]
    motor_cmd.id(i); motor_cmd.q(0.0); motor_cmd.dq(0.0); motor_cmd.tau(0.0) # defaults
    if i == 30:      # Neck Pitch
        motor_cmd.q(0.3); motor_cmd.kp(5); motor_cmd.kd(0.1)
    elif i == 29:
        motor_cmd.kp(2); motor_cmd.kd(0.05)
    else:
        motor_cmd.kp(0.0); motor_cmd.kd(0.0)

# kinematic_modes is a 5-element array by parallel-link group
# (waist/L_ankle/R_ankle/L_wrist/R_wrist).
# (C++ example: cmd.kinematic_modes().fill(KinematicMode.PJS))
lowcmd_publisher.write(low_cmd_msg)
```

"The full example can be found in the GitHub repository."

Camera

1. IGRIS-C Camera Topic List

- Data Topic - `igris_c::msg::dds::CompressedMessage`

TopicName	Description
<code>igris_c/sensor/d435_color</code>	RealSense D435 RGB image data
<code>igris_c/sensor/d435_depth</code>	RealSense D435 depth image data
<code>igris_c/sensor/eyes_stereo</code>	IGRIS-C head stereo camera image data
<code>igris_c/sensor/left_hand</code>	IGRIS-C left hand end-effector image data
<code>igris_c/sensor/right_hand</code>	IGRIS-C right hand end-effector image data

- Service

TopicName	Message Type	Description
<code>igris_c/sensor/realsense_config_request</code>	<code>igris_c::msg::dds::RealsenseConfigRequest</code>	RealSense D435 configuration service
<code>igris_c/sensor/stereo_config_request</code>	<code>igris_c::msg::dds::StereoConfigRequest</code>	IGRIS-C head stereo camera configuration service
<code>igris_c/sensor/mono_config_request</code>	<code>igris_c::msg::dds::MonoConfigRequest</code>	IGRIS-C hand end-effector camera configuration service
<code>igris_c/sensor/camera_config_response</code>	<code>igris_c::msg::dds::ServiceResponse</code>	Configuration service response

2. Camera Topic & Service API

- SDK Initialization

```

#include <igris_sdk/channel_factory.hpp>
#include <igris_sdk/igris_c_client.hpp>
#include <igris_sdk/igris_c_msgs.hpp>
#include <igris_sdk/subscriber.hpp>

using namespace igris_sdk;
using CompressedMessage = igris_c::msg::dds::CompressedMessage;

int main() {
    // Initialize SDK with domain ID
    int domain_id = 0;
    ChannelFactory::Instance()->Init(domain_id);
    if (!ChannelFactory::Instance()->IsInitialized()) {
        std::cerr << "Failed to initialize ChannelFactory" << std::endl;
        return 1;
    }

    // Initialize service client
    Igrisc_Client client;
    client.Init();
    client.SetTimeout(10.0f); // Set the default timeout to 10 seconds

    // Service calls...
}

```

- Image Subscriber Example

```

Subscriber<CompressedMessage> sub("igris_c/sensor/d435_color");

sub.init([](const CompressedMessage &msg) {
    cv::Mat raw(1, msg.image_data().size(), CV_8UC1,
               const_cast<uint8_t*>(msg.image_data().data()));

    cv::Mat img = cv::imdecode(raw, cv::IMREAD_UNCHANGED);
});

```

- JPEG-compressed image transmission

- Image Subscriber Example

```
#include <dds/dds.hpp>
#include <igrid_sdk/igrid_c_msgs.hpp>

#include <chrono>
#include <iostream>
#include <thread>

using namespace igrid_c::msg::dds;

int main() {
    int domain_id = 10;

    // Create DDS participant
    dds::domain::DomainParticipant participant(domain_id);
    dds::pub::Publisher publisher(participant);
    dds::sub::Subscriber subscriber(participant);

    // Define topics
    dds::topic::Topic<StereoConfigRequest> req_topic(
        participant, "igrid_c/sensor/stereo_config_request");

    dds::topic::Topic<ServiceResponse> res_topic(
        participant, "igrid_c/sensor/camera_config_response");

    // Create writer and reader
    dds::pub::DataWriter<StereoConfigRequest> writer(publisher, req_topic);
    dds::sub::DataReader<ServiceResponse> reader(subscriber, res_topic);

    // Create request ID
    std::string request_id = "req_001";

    // =====
    // Build request
    // =====
    StereoConfigRequest req;
    req.request_id(request_id);
    req.stereo_width(1280);
    req.stereo_height(480);
    req.stereo_fps(30);
    req.stereo_jpeg_quality(90);
    req.enable_stereo(true);
```

```

// publish
writer.write(req);
std::cout << "[publish] StereoConfigRequest sent\n";

// =====
// Wait for response
// =====
auto start = std::chrono::steady_clock::now();
int timeout_ms = 5000;

while (true) {
    auto samples = reader.take();

    for (auto &s : samples) {
        if (!s.info().valid()) continue;

        const auto &res = s.data();

        if (res.request_id() == request_id) {
            std::cout << "\n=== Response ===\n";
            std::cout << "success: " << res.success() << "\n";
            std::cout << "message: " << res.message() << "\n";
            std::cout << "error_code: " << res.error_code() << "\n";
            return 0;
        }
    }

    // Check timeout
    auto now = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
start).count();
    if (elapsed > timeout_ms) {
        std::cout << "[timeout] no response\n";
        break;
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(50));
}

return 0;
}

```

* Examples are available in the GitHub repository:

- https://github.com/robrosinc/igris_c_sdk_public
 - [Image data subscription](#) in C++
 - [Image data subscription](#) in Python
 - [Camera configuration service](#) in C++

Hand

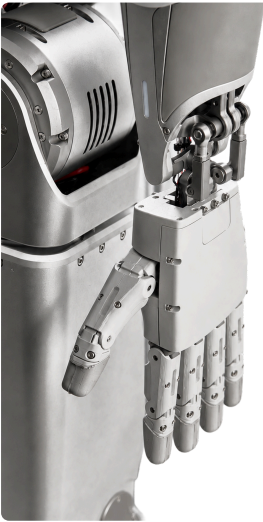
1. End Effector Overview

IGRIS-C supports several types of end effectors depending on the application. Regardless of type, all share the same DDS control interface ([rt/handcmd](#) / [rt/handstate](#) / [rt/service/hand_init](#)); only the number of drive motors, motor IDs, and communication speed differ by type.

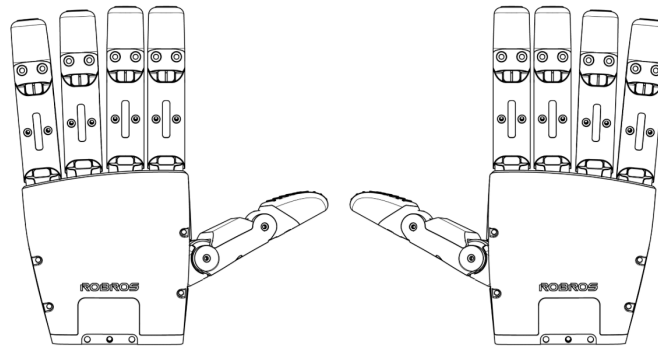
Type	Description	Motor IDs	Baud
Dexterous Hand	Five-finger hand (left/right)	R 11-16, L 21-26 (12 motors)	1 Mbps
1-DoF	Single-degree-of-freedom gripper	R 11, L 21	1 Mbps
Magnet	Electromagnet end effector	R 11-12, L 21-22 (4 motors)	57600 bps

“The end effector type is preconfigured at shipping based on the robot's configuration. The control interface described in this document applies identically regardless of type.”

Appearance by type

Dexterous Hand	1-DoF	Magnet
		

2. Dexterous Hand



- The thumb has **2 independently driven degrees of freedom**, while the remaining four fingers each operate with **1 coupled drive degree of freedom**. All fingers additionally have **1 DoF for passive adaptation**, providing grasping performance that naturally conforms to object shape.
- The hand is divided into **left/right hands** with a symmetric structure. Left/right are identified by different motor ID ranges (R 11–16, L 21–26), and are **controlled together based on ID over a single DDS channel** (`rt/handcmd` / `rt/handstate`).
- The 1-DoF gripper drives 2 motors, R(11)/L(21); the magnet drives 4 motors, R(11·12)/L(21·22) (11/21 = magnet on/off, 12/22 = position servo), and both are controlled by their respective IDs over the same `rt/handcmd` channel.

3. IGRIS-C Hand Node Usage Guide

Node Overview

`igris_c_hand_node` is a C++ node that allows the IGRIS-C end effector to be controlled entirely through `igris_c_sdk` DDS channels. It performs initialization, target pose transmission, and status reception using only the `HandCmd` / `HandState` / `HandInitRequest` / `ServiceResponse` messages. By default, the node uses the `/dev/igris_c_hand` port.

Subscribe / Publish Interface

The node communicates via `igris_c_sdk` DDS channels, not standard ROS2 topics.

Channels the node subscribes to

1. `rt/handcmd` (`igris_c_sdk::HandCmd`) — **Target position command**. `motor_cmd` is a variable-length sequence of `MotorCmd`, with each entry in `(id, q, ...)` format.
 - For entries whose `id` is one of the valid IDs (hand: {11–16, 21–26}, 1-DoF: {11, 21}, magnet: {11, 12, 21, 22}), the `q` value is stored as a normalized (0.0–1.0) target for that joint and applied immediately. Out-of-range values are clamped, and IDs not in the mapping are ignored.

- **▲ Initialization is no longer handled via `rt/handcmd`** . (The former `id=99` trigger method has been removed.) Use the `rt/service/hand_init` service below for initialization.

Channels the node publishes

1. `rt/handstate` (`igris_c_sdk::HandState`) — Publishes an array of measured motor states at a fixed interval. Each `MotorState` field:
 - `q` : Current position (0–1)
 - `dq` : Current velocity
 - `tau_est` : Estimated current value (the hand carries motor current in this field)
 - `temperature` : Motor temperature (°C)
 - `status_bits` : Motor error/status bits

Initialization (Calibration) Service — `rt/service/hand_init`

Initialization is handled as a separate **Request/Response service**.

- **Request:** Publish `HandInitRequest { Header header; string request_id }` to `rt/service/hand_init/request` .
- **Response:** `ServiceResponse { request_id, success, message, error_code }` is returned on `rt/service/hand_init/response` (`error_code == 0` indicates success). The request and response are matched by `request_id` .
- Upon receiving the request, the node resets the targets to 0 and performs `effector->initialize()` (type-specific calibration/homing).
- **High-level convenience method:** Calling the SDK's `Igrisc_Client::InitHand(timeout_ms)` handles the above request/response internally. The e.4 example below uses the lower-level approach of handling the channel directly.

Sequence Example

1. The node may auto-initialize at startup depending on the robot configuration. If reinitialization is needed at runtime, publish a `HandInitRequest` (with `request_id` filled in) to `rt/service/hand_init/request` .
2. Publish target positions by populating `motor_cmd` on `rt/handcmd` . E.g., only the right index finger to 0.8 → `[{id:12, q:0.8}]` ; the same pose for both hands → publish a sequence including all 12 IDs.
3. Status check: subscribe to `rt/handstate` and monitor the `motor_state` array.
4. Initialization result check: subscribe to `rt/service/hand_init/response` and check `success` / `error_code` .

Debugging Tips

- `HandState.status_bits` contains per-motor fault/limit bits, so use it for alerts/status display in a GUI or logger.
- `HandCmd.motor_cmd[].q` is a normalized value from 0–1. If scaling is needed to match the hand's structure, apply a linear transformation in the upper-level application.
- If origin calibration is suspected to have drifted, request init again via `rt/service/hand_init/request` to recalibrate. Check the response on `rt/service/hand_init/response`.

4. IGRIS-C Hand Example

"Full runnable example: `examples/cyclonedds/cyclonedds_hand_example.cpp` (C++), `examples/python/hand_example.py` (Python). Below is an excerpt of the core flow; topics are resolved under the robot unit namespace as described in b.3."

```

// Example: drive the igris_c_hand node entirely through igris_c_sdk DDS channels.
// * Trigger initialization via rt/service/hand_init/request (HandInitRequest)
// * Send target vectors via rt/handcmd (MotorCmd.id + q)
// * React to rt/handstate and rt/service/hand_init/response for logging/verification
//
// Build against igris_c_sdk and run after sourcing the workspace.

#include <algorithm>
#include <array>
#include <chrono>
#include <thread>

#include <igris_c_sdk/channel_factory.hpp>
#include <igris_c_sdk/publisher.hpp>
#include <igris_c_sdk/subscriber.hpp>
#include <igris_c_sdk/types.hpp>

// types.hpp aliases HandCmd / HandState / MotorCmd / HandInitRequest /
// ServiceResponse and the
// infra (ChannelFactory / Publisher / Subscriber / QosProfile) into the igris_c_sdk
// namespace.
using namespace igris_c_sdk;

static constexpr std::array<uint16_t, 12> kMotorIds = {
    11, 12, 13, 14, 15, 16, 21, 22, 23, 24, 25, 26};

static HandCmd make_targets(const std::array<float, 12>& q) {
    HandCmd cmd;
    for (size_t i = 0; i < kMotorIds.size(); ++i) {
        MotorCmd m;
        m.id(kMotorIds[i]);
        m.q(std::clamp(q[i], 0.0f, 1.0f));
        cmd.motor_cmd().push_back(m);
    }
    return cmd;
}

int main() {
    ChannelFactory::Instance()->Init(0, "igris_c_IG01"); // domain 0, robot unit namespace
    (see b.3)

    Publisher<HandCmd> cmd_pub("rt/handcmd", QosProfile::SensorData());

```

```

cmd_pub.init();

// Initialization is a request/response service (no more id=99 in rt/handcmd).
Publisher<HandInitRequest> init_req_pub("rt/service/hand_init/request",
QosProfile::Services());
init_req_pub.init();

Subscriber<HandState> state_sub("rt/handstate", QosProfile::SensorData());
state_sub.init([](const HandState& s) {
    if (s.motor_state().size() >= 3) {
        printf("q[0..2]=%.3f %.3f %.3f\n",
            s.motor_state()[0].q(),
            s.motor_state()[1].q(),
            s.motor_state()[2].q());
    }
});

Subscriber<ServiceResponse> init_res_sub(
    "rt/service/hand_init/response", QosProfile::Services());
init_res_sub.init([](const ServiceResponse& r) {
    printf("[hand_init] success=%d code=%d msg=%s\n",
        r.success(), r.error_code(), r.message().c_str());
});

// 1) Trigger initialization via the hand_init service
HandInitRequest init_req;
init_req.request_id() = "hand_init";
init_req_pub.write(init_req);
std::this_thread::sleep_for(std::chrono::milliseconds(1500));

// 2) Fully open
cmd_pub.write(make_targets({0.05f, 0.05f, 0.05f, 0.05f, 0.05f, 0.05f,
    0.05f, 0.05f, 0.05f, 0.05f, 0.05f, 0.05f}));
std::this_thread::sleep_for(std::chrono::seconds(2));

// 3) Close all joints
cmd_pub.write(make_targets({0.85f, 0.85f, 0.85f, 0.85f, 0.85f, 0.85f,
    0.85f, 0.85f, 0.85f, 0.85f, 0.85f, 0.85f}));
std::this_thread::sleep_for(std::chrono::milliseconds(2500));

// 4) Basic pinch (right thumb + index + middle), others relaxed
std::array<float, 12> pinch = {};
pinch[0] = 0.80f; // right thumb (id=11)

```

```
pinch[1] = 0.75f; // right index (id=12)
pinch[2] = 0.60f; // right middle (id=13)
cmd_pub.write(make_targets(pinch));
std::this_thread::sleep_for(std::chrono::seconds(2));

// 5) Back to open
cmd_pub.write(make_targets({0.05f, 0.05f, 0.05f, 0.05f, 0.05f, 0.05f,
                           0.05f, 0.05f, 0.05f, 0.05f, 0.05f}));
std::this_thread::sleep_for(std::chrono::milliseconds(1500));

cmd_pub.stop();
state_sub.stop();
init_res_sub.stop();
init_req_pub.stop();
return 0;
}
```