密　　级：公开
文件编号：UM-XH-M1-V1.0

X-Hand M1 SDK 使用手册

北京达奇月泉仿生科技有限公司

V1.0

目录

前言

1.1 编写目的

本文旨在帮助「普通用户/科研人员/运维人员」快速熟悉「X-Hand M1 SDK」的核心功能、操作流程及常见问题处理方法，降低使用门槛，确保软件高效稳定运行。

1.2 适用范围

本手册适用于信手 M1 (X-Hand M1) 配套 SDK V1.0 版本

1. 软件概述

1.1. 软件定位与核心价值

X-Hand M1 SDK (Software Development Kit) 是面向 M1 版本信手 (X-Hand M1) 用户的核心开发工具包, 是机器手硬件与上层应用之间的桥梁。它封装了机器手运动控制、IO 交互等底层核心功能，提供了一套标准化 API 接口、开发文档及示例代码，支持多语言(C++、Python)适配。

SDK 旨在降低机器手使用和二次开发门槛, 让开发者无须深入理解底层硬件驱动与运动学算法，即可快速实现机器手与分拣系统、视觉检测、产线 MES 系统的集成开发，广泛适用于 3C 电子、汽车制造、仓储物流、医疗器械等行业的自动化产线定制场景。

SDK 同步适配机器人操作系统(ROS)生态，专门提供基于 ROS 2(Jazzy LTS 版本)的标准化功能包，该功能包将信手 SDK 的核心控制能力(运动控制、状态反馈等)完整封装为独立的 ROS 2 节点，并遵循 ROS 2 通信规范设计接口:

- 对外以 ROS Topic 的形式开放控制指令下发，同时提供状态反馈 Topic；
- 对内通过 SDK 原生 API 与机器手硬件建立低延迟通信，实现指令的解析、转发与状态的实时回传。

为了区分，下文以「SDK」表示 C++ SDK 本身，以「SDK-ROS」表示带 ROS 框架的 SDK

1.2. 运行环境要求

硬件环境要求:

四核及以上处理器（x86 架构，Intel Core i5 及以上，AMD Ryzen 5 及以上），主频 2.5GHz+；最低 4GB DDR4 2400MHz 内存，推荐 8GB 及以上；最低 10GB 可用存储空间（含 SDK 代码库、示例项目及日志存储），推荐使用 SSD 硬盘；需配备以太网接口(100Mbps)，可选配 CAN 接口(CAN Classic)。

软件环境要求:

SDK 基于 64 位 Linux 系统，支持 Ubuntu 20.04/22.04/24.04，不同版本依赖库已集成，无须额外配置。SDK-ROS 基于 Ubuntu24.04，ROS2 jazzy。

1.3. 安装与配置

SDK 在 Ubuntu 下安装编译器和 cmake 工具即可使用。可直接安装 build-essential 包：

```
username@hostname:~$ sudo apt update
...
username@hostname:~$ sudo apt install build-essential -y
```

验证编译器是否安装成功 (示例基于 Ubuntu 24.04)：

```
username@hostname:~$ gcc --version
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

如上，成功后会输出 gcc 版本信息。

验证 make 工具(示例基于 Ubuntu 24.04)：

```
username@hostname:~$ make --version
GNU Make 4.3
为 x86_64-pc-linux-gnu 编译
Copyright (C) 1988-2020 Free Software Foundation, Inc.
许可证：GPLv3+：GNU 通用公共许可证第 3 版或更新版本
<http://gnu.org/licenses/gpl.html>。
本软件是自由软件：您可以自由修改和重新发布它。
在法律允许的范围内没有其他保证。
```

如上，成功后会输出 make 版本信息。

SDK-ROS 在此基础上还需要安装 ROS2 Jazzy，安装方法可参考：

Ubuntu 24.04 安装 ROS 2 Jazzy 完整指南 | Zed

如何在 ubuntu24.04 安装 jazzy | 鱼香 ROS

2. 快速入门

SDK 提供了两种运行模式：作为 ROS 的一个节点嵌入到 ROS 框架中运行(参考 2.5 节、2.6 节)；单独运行 SDK，在 C++中编写控制 Demo(参考 2.3 节、2.4 节)。

2.1 节、2.2 节作为运行 SDK 的前置条件，简要介绍了硬件连接和 PC 侧网络 IP 设置。

2.1. 硬件连接

硬件连接包含两个部分：供电和通信（网线）。

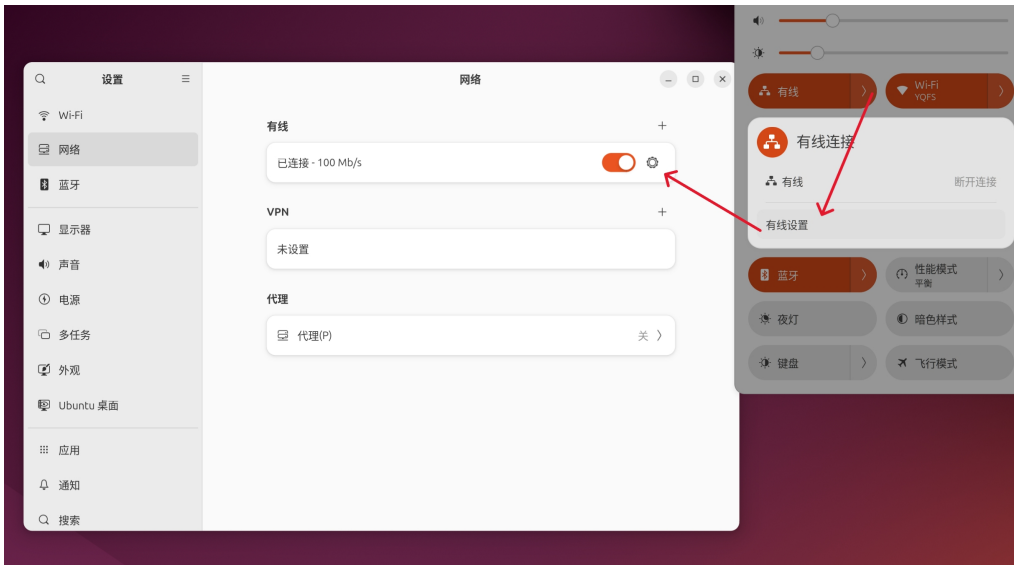
如果驱动单只信手，推荐程控电源 24V~48V / 3A 供电；信手引出的网线直接连接运行 SDK 的 PC 机 / mini PC / 开发板。

信手主控初始化时间在 1s 左右，建议信手上电 1s 后运行 SDK 可执行文件。

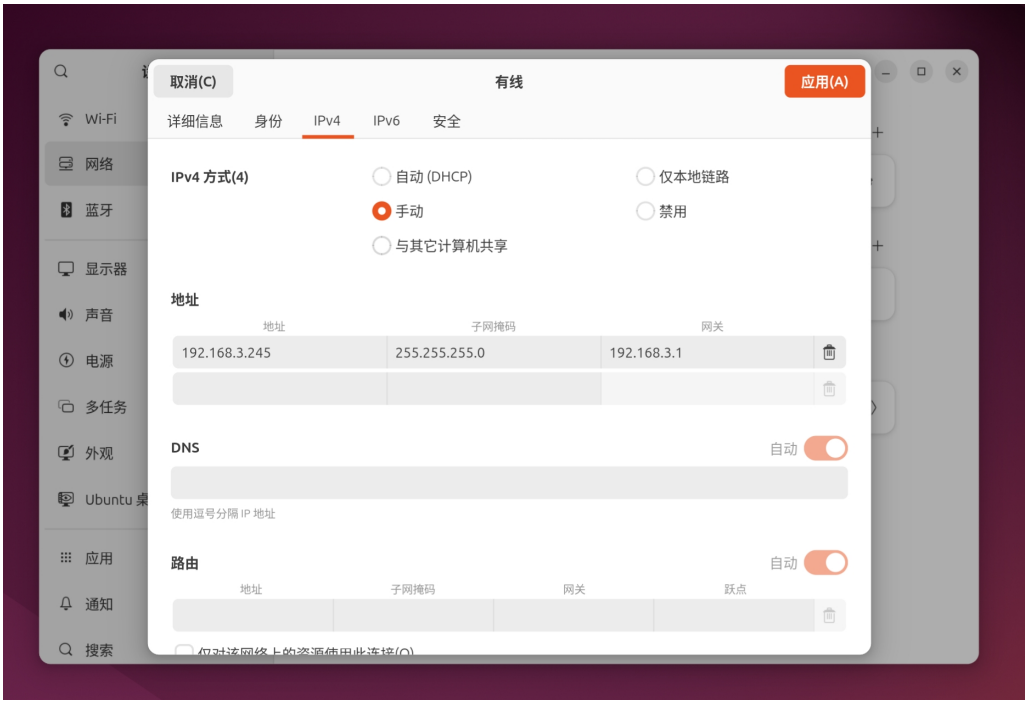
2.2. PC 端网络设置

PC 端运行的 SDK 与 X-Hand 间通过 UDP 通信，需要配置 PC 端有线连接 IP。

如下，打开网络设置（需要连接 X-Hand 并上电）：



将 IP 配置为手动，IP 地址设为 192.168.3.245，具体如下：



2.3. SDK 编译

SDK 下载地址：

SDK 通过自带工具 Tool.sh 脚本（sdk_2 目录下）编译。

1) 开启 Tool.sh 参数自动补全

Tool.sh 有参数自动补全功能，建议首次使用时先开启。

先 cd 到 Tool.sh 所在目录(示例中~/.../sdk_2_out)，然后执行命令

```
./Tool.sh completion
```

```
username@hostname:~/.../sdk_2_out$  ./Tool.sh completion
===== 配置自动补全 =====
添加补全配置到 /home/username/.bashrc...
清理旧的补全绑定...
清理旧的补全配置...
☑ 自动补全配置已写入 /home/username/.bashrc!
请执行以下命令使配置立即生效：
 source /home/username/.bashrc
或新开一个终端会话。
```

注意，首次使用时，completion 参数是不会自动补全的，需要手动输入。执行后需

要根据打印提示新开一个终端或执行 `source /home/username/.bashrc` 命令，其中 username 是用户名，`./Tool.sh completion` 命令的打印会自动根据实际用户名填充而无须手动修改，直接复制这一行执行即可。

2) 编译

Tool.sh 所在目录下执行命令 (下面 make 命令的打印示例只展示首尾部分内容)

```
./Tool.sh make
```

```
username@hostname:~/.../sdk_2_out$  ./Tool.sh make
===== 开始构建项目 =====

无须执行第三方编译(lib 已存在且未指定 make 参数)
-- Project root: /home/username/workspace/sdk_2_out
-- ===== PROJECT STRUCTURE =====
-- PROJECT_ROOT: /home/username/workspace/sdk_2_out
-- SRC_DIR: /home/username/workspace/sdk_2_out/src
-- INC_DIR: /home/username/workspace/sdk_2_out/inc
-- LIB_DIR: /home/username/workspace/sdk_2_out/lib/x86_64
-- THIRD_PARTY_DIR: /home/username/workspace/sdk_2_out/
third_party
-- ============================
-- ===== CONFIGURATION SUMMARY =====
-- Found modules:
--   [Auto_Set_Id] -> BUILD_MODULE_AUTO_SET_ID = OFF

... [中间过程省略]

[ 77%] Building CXX object src/Switch_Board/CMakeFiles/Switch_B
oard.dir/Timer.cpp.o
[ 81%] Building CXX object src/Switch_Board/CMakeFiles/Switch_B
oard.dir/Serial.cpp.o
[ 85%] Building CXX object src/Switch_Board/CMakeFiles/Switch_B
oard.dir/GPIO.cpp.o
[ 88%] Linking CXX shared library /home/username/workspace/sdk
_2/lib/x86_64/libSwitch_Board.so
[100%] Built target Switch_Board

===== 项目构建完成 =====
```

3) 清理

Tool.sh 所在目录下执行命令

```
./Tool.sh clean
```

```
username@hostname:~/.../sdk_2_out$  ./Tool.sh clean
===== 开始清理项目 =====
删除 build 目录...
===== 项目清理完成 =====
```

2.4. SDK 运行

完成 IP 设置、连接好硬件、上电后，在 Ubuntu 终端 cd 到 sdk_2/bin 目录下，执行编译生成的可执行文件即可运行 SDK。

2.5. SDK-ROS 编译

本节默认 PC 侧已安装 ROS2 Jazzy。

SDK ROS 下载地址：

可直接下载代码压缩包，或者用 git clone 到本地

```
git clone --recurse-submodules git@gitee.com:beijing-daqi-yue
quan-bionics/sdk2ros.git
```

因为 SDK 本身是 SDK-ROS 的一个子仓，所以需要同时拉取子仓代码：

```
git submodule update --remote --merge
```

cd 到 sdk2ros/目录下，执行

```
colcon build
```

SDK-ROS 会自动调用 SDK 的.so 库, clone 下来的 SDK 子仓中已包含所需.so 文件，无需提前手动编译 SDK。如果 SDK 部分有改动，则需要先编译 SDK(参考 2.5 节)，再执行 `colcon build` 命令编译 SDK-ROS。

没有与 colcon build 对应的 clean 命令，手动删除 sdk2ros/下的 build/, install/, log/, logs/ 目录即可.

2.6. SDK-ROS 运行

编译生成的可执行文件路径：*sdk2ros \ install \ devices_pkg \ lib \ devices_pkg*

SDK-ROS 节点支持两种启动方式：ros2 run 和 launch 启动。

1. ros run 启动

首先在当前 shell 执行配置脚本

```
source ./install/setup.sh
```

然后执行可执行文件

```
ros2 run devices_pkg「可执行文件名」
```

2. launch 文件启动

SDK-ROS launch 文件路径：*sdk2ros \ src \ devices_pkg \ launch \ launch.py*

```
def generate_launch_description():
    return LaunchDescription([
     Node(
            package='devices_pkg',
            executable='Test_Node',# CMakeLists.txt 中定义的可执行文件名
            name='test_node',       # 节点运行时的名称
            output='screen',        # 将节点的输出打印到屏幕
           #launch 退出时关闭节点
            on_exit=Shutdown(reason="launch is shutting down")
        ),
     Node(
            package='devices_pkg',
            executable='X_Hand_Node',
            name='x_hand_node',
            on_exit=Shutdown(reason="launch is shutting down")
        )
    ])
```

SDK-ROS 默认 launch 文件中包含两个节点：Test_Node 是测试节点，定时发送控制命令控制信手握拳/张开；X_Hand_Node 是信手控制节点。（详见 3.4 节）

若用户需要用自己的节点控制信手，需要在此 launch 文件中把 Test_Node 节点注释掉，防止命令冲突。

3. 功能模块详解

3.1. SDK 主要目录结构

SDK 主要目录结构如下:

```
sdk_2_out
├── bin/                                  // Tool.sh 工具编译生成的可执行文件
├── config/                               // cmake、OTA 包、docker 镜像等
│   └── YAML/                             // Yaml 文件，用于配置设备参数
├── DOC/                                  // SDK 相关文档
├── inc/                                  // SDK 代码头文件
├── lib/                                  // SDK 涉及的库文件 (支持 x86+ARM)
├── script/                               // SDK 内部使用的处理脚本
├── src/                                  // SDK 代码源文件
│   └── Project/
│       └── X-Hand/
│           └── out/
│               └── X_Hand_Out.cpp        // 用户可在此编写手部运动控制 Demo
├── third_party/                          // SDK 涉及的第三方库
├── CMakeLists.txt                        // CMake 主配置文件
└── Tool.sh                               // SDK 操作脚本
```

3.2. SDK-ROS 主要目录结构

```
sdk2ros
├── build/                                // SDK-ROS 编译生成
├── install/                              // SDK-ROS 编译生成，可执行文件
├── log/                                  // SDK-ROS 编译生成，日志文件
├── logs/                                 // SDK-ROS 编译生成，日志文件
├── src/
│   └── devices_pkg/
│       ├── src/
```

```
|       |   ├── project/          // SDK-ROS 代码
|       |   └── sdk/              // SDK 代码（参考 3.2 节）
|       └── msg/                  // ROS2 Topic msg 定义
└── CMakeList.txt                 // SDK-ROS Cmake 配置文件
```

3.3. 信手控制框架解析(SDK 部分)

信手控制 Demo 框架路径：sdk_2_out \ src \ Project \ X-Hand \ out \ X_Hand_Out.cpp
通过 SDK 内部接口直接控制信手电机，以完成不同动作(直驱电机，不含运控)。

设备/控制指针定义

```
// 硬件设备指针，SDK 中用于总揽所有硬件设备
Robot_Hardware* X_Hand;

// 电机设备指针，对应 X-Hand 6 台电机
shared_ptr<Device_Struct> Motor_1_D;
shared_ptr<Device_Struct> Motor_2_D;
shared_ptr<Device_Struct> Motor_3_D;
shared_ptr<Device_Struct> Motor_4_D;
shared_ptr<Device_Struct> Motor_5_D;
shared_ptr<Device_Struct> Motor_6_D;

// X-Hand 触觉传感器设备指针
shared_ptr<Device_Struct> Tactile_Sensor_D;

// 电机控制对象指针，对应 X-Hand 6 台电机
Motor* Motor_1_Control;
Motor* Motor_2_Control;
Motor* Motor_3_Control;
Motor* Motor_4_Control;
Motor* Motor_5_Control;
Motor* Motor_6_Control;

// 触觉传感器控制对象指针
Hw_Pressure_Sensor* Tactile_Sensor_Control;
```

SDK 兼容 ROS2 框架（详见 2.5、2.6、3.2、3.4 节），为方便 ROS 调用，设备指针均定义为全局变量。

如上，除 Robot_Hardware 指针外，设备指针包含两类：设备指针、控制对象指针。

- **设备指针**

  用于 SDK 内部接口中获取设备通信通道、UDP 通信 ID、UDP 通信 cmd 以及调用 UDP Send 接口(UDP 通信接口暂不开放给用户);

- **控制对象指针**

  用于调用设备控制接口，用于具体设备控制操作。

预设结构、变量定义

```
// X-Hand 反馈数据结构
typedef struct X_hand_FB {
   float P;
   float V;
   float F;
   float temp[2];
   u16 error;
} X_hand_FB;

// X-Hand 发送数据结构，用于向 X-Hand 发送控制指令
typedef struct X_hand_Send_Data {
   float P;
   float V;
   float F;
   float KP;
   float KD;
} X_hand_Send_Data;

// 用于 X-Hand 初始化
int init_time_step = 1000 * 1000 / 500;
float Pos_Offest[6] = {0, 0, 0, 0, 0, 0};
float Motor_Mirror[6] = {-1, -1, 1, 1, 1, 1};
// float Motor_K[6] = {1, 1, 1, 1, 1, 1};
float Motor_K[6] = {3103 - 20, 421.953 - 5, 3779.49 - 20, 3833.81 - 20,
                    3739.36 - 20, 3780.85 - 20};
// X-Hand 反馈/发送数据结构数组
X_hand_Send_Data Send_Datas[6];
X_hand_FB FB_Datas[6];
// 用于 X-Hand 传感器数据存储
map<u8, vector<u16>> g_sensor_data;
```

SDK 预设变量/结构如上，其中用于 X-Hand 初始化的部分不需要用户关注，保持不变即可。

X_hand_Send_Data 结构

用于构建向信手发送的控制指令的结构。

- P – 电机目标位置
- V – 电机目标速度
- F – 电机目标扭矩
- KP – 电机 MIT 模式 Kp 参数
- KD – 电机 MIT 模式 Kp 参数

X_hand_FB 结构

用于接收信手的反馈数据，内容类似。

- P – 当前电机位置
- V – 当前电机速度
- F – 当前电机扭矩
- temp – 保留字段
- Error – 错误码

g_sensor_data

触觉传感器返回数据。是一个 u8 和一个 vector<u16>的 map，具体定义如下：

- U8 – 触觉传感器 ID
- vector<u16> – 触觉传感器返回的点阵数据

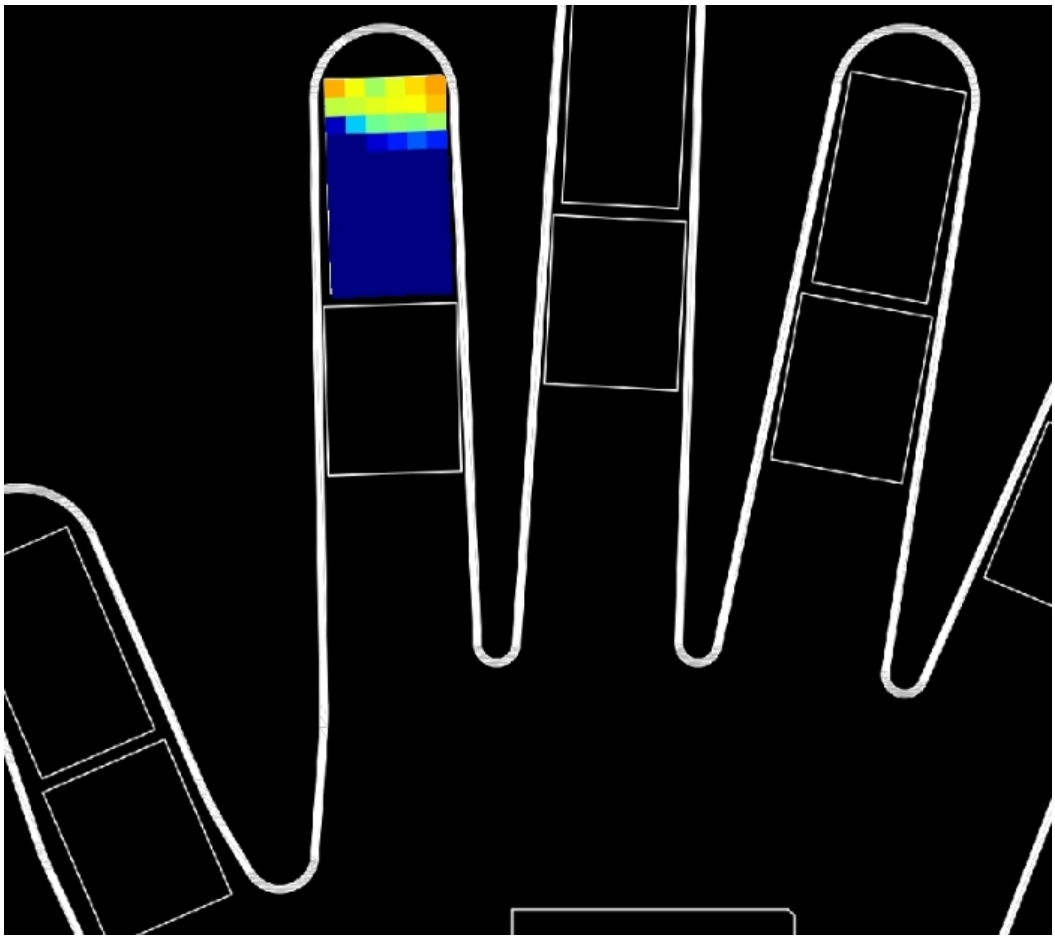
触觉传感器传回的是点阵数据，手指为 6*12 点阵，手掌处为 8*14 点阵。每个点范围 0~4095，无符号整数，**非标量**，与该点的实际压力大小对应，但**非线性关系**。

触觉传感器传回的是点阵数据的一维形式，即按 6 行 12 列或 8 行 14 列形式逐行扫描，依次传回，存入 vector<u16>中。

信手不同位置的触觉传感器 ID 如下：

触觉传感器 ID	传感器安装位置
0x01	拇指
0x02	食指
0x03	中指
0x04	无名指
0x05	小指
0x06	掌心

触觉传感器均为矩形，正向安装，以手指为例，从左到右为 1~8，从上到下为 1~12：



如上，彩色部分即触觉传感器(左手食指为例)。传感器返回的数据如下图所示：

数值

	1	2	3	4	5	6
1	2864	2505	2184	2466	2682	2909
2	2279	2336	2431	2501	2529	2787
3	274	1330	1971	2078	2025	2153
4	0	0	334	616	856	646
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0

手指传感器为 6*12 点阵，传回 vector<u16>的数据依次为上图从左至右，从上至下线性排列([1][1], [1][2], [1][3]... [1][6], [2][1], [2][2], ... [2][6], ... [12][1], [12][2], ... [12][6])。传感器数据仅作为该点压力大小的定性分析使用。

手掌传感器与手指处安装方向一致，仅行列点数不同(8*14)。

手指、手掌传感器在左右手安装方向均一致，即假定观察者面向掌心，五指完全张开时指向为上，则传感器行号由上至下依次增大，列号由左至右依次增大。

X-Hand 初始化函数

```
// X-Hand 初始化函数涉及的接口
void Get_FB(void) {...}
void Send(void) {...}
void Get_K(void) {...}

// X-Hand 初始化
void X_hand_Init(void) {...}
// 触觉传感器初始化
void Tactile_Sensor_Init(void) {...}
```

如上（函数实现未列出），上述函数用户无须关注，也不要修改，否则信手初始化异常，可能导致信手控制异常，严重则会损坏信手硬件。

SDK main 函数

此部分为 SDK 控制核心代码，用户可在 main 中加入电机、触觉传感器设备控制代码，以进行设备简单控制 Demo。

```
#ifndef HAVE_ROS
int main(int argc, char* argv[])
#else
int hardware_init(string ADDR)
#endif
{
  // 新建机器人硬件对象
  X_Hand = new Robot_Hardware();

  // 根据 yaml 文件配置初始化设备对象
  X_Hand->Add_Device_Type("Switch_Board", Switch_Board_Device_Init,
```

```
                                                Switch_Board_Device_CallBack_F,
                                                Switch_Board_Device_Delete_F);
    X_Hand->Add_Device_Type("Motor_Device", Motor_Device_Init,
                                                Motor_Device_CallBack_F,
                                                Motor_Device_Delete_F);

    // Auto_Set_Id_Type 用户无须关心，是信手内部规划设备 ID 功能抽象成了设备，
       硬件设备实际不存在
     X_Hand->Add_Device_Type(Auto_Set_Id_Type, Auto_Set_Id_Init,
                                Auto_Set_Id_CallBack_F, Auto_Set_Id_Delete_F);
    X_Hand->Add_Device_Type("Tactile_Sensor_Custom", Hw_Pressure_Sensor_Init,
                                Hw_Pressure_Sensor_CallBack_F,
                                Hw_Pressure_Sensor_Delete_F);

#ifndef HAVE_ROS
    // 设置可执行文件路径和 yaml 文件路径，用户无须关心
    filesystem::path exe_path = filesystem::canonical("/proc/self/exe");
    filesystem::path dir_path = exe_path.parent_path();
    string ADDR = dir_path.string() + "/../config/YAML/X_Hand/out/TOP.yaml";
#endif
    // SDK 初始化 X-Hand 设备，固定操作，用户无须关心，不要修改
    if (X_Hand->Init_TOP(ADDR) != 0) {
        cout << "Init_ERR" << endl;
        return -1;
    }

    // 获取设备对象，用户无须关心，不要修改
    Motor_1_D = X_Hand->Get_Device_For_Name("Motor_1");
    Motor_2_D = X_Hand->Get_Device_For_Name("Motor_2");
    Motor_3_D = X_Hand->Get_Device_For_Name("Motor_3");
    Motor_4_D = X_Hand->Get_Device_For_Name("Motor_4");
    Motor_5_D = X_Hand->Get_Device_For_Name("Motor_5");
    Motor_6_D = X_Hand->Get_Device_For_Name("Motor_6");
    Tactile_Sensor_D = X_Hand->Get_Device_For_Name("Tactile_Sensor");

    // 获取设备控制对象，用户无须关心，不要修改
    Motor_1_Control = (Motor*)X_Hand->Get_Control_Class(Motor_1_D);
    Motor_2_Control = (Motor*)X_Hand->Get_Control_Class(Motor_2_D);
    Motor_3_Control = (Motor*)X_Hand->Get_Control_Class(Motor_3_D);
    Motor_4_Control = (Motor*)X_Hand->Get_Control_Class(Motor_4_D);
    Motor_5_Control = (Motor*)X_Hand->Get_Control_Class(Motor_5_D);
    Motor_6_Control = (Motor*)X_Hand->Get_Control_Class(Motor_6_D);
```

```
    Tactile_Sensor_Control =
     (Hw_Pressure_Sensor*)X_Hand->Get_Control_Class(Tactile_Sensor_D);

    // 初始化 X-Hand 设备，固定操作，用户无须关心，不要修改
    X_hand_Init();
    // 初始化传感器设备，固定操作，用户无须关心，不要修改
    Tactile_Sensor_Init();

   /* ----------
    用户可在此添加控制代码
    ---------- */

    // 获取信手反馈数据
#ifndef HAVE_ROS
    float loop_time_step = 1000 * 1000 / 500;
    int times = 0;
    float test = 0.5;
    while (1) {
        if (times % 500 * 2 == 0) {
            test = -test;
        }
        times++;
        for (int i = 0; i < 6; i++) {
            Send_Datas[i].P = 0.5 + test;
            Send_Datas[i].V = 0;
            Send_Datas[i].F = 0;
            Send_Datas[i].KP = 500;
            Send_Datas[i].KD = 5;
        }
        Send_Datas[0].P = 0.25 + test / 2;
        Send_Datas[1].P = 0.1;
        Send();
        usleep(loop_time_step);
        Get_FB();

        for (int i = 0; i < 6; i++) {
            cout << "num: " << i << " " << FB_Datas[i].P << endl;
        }
    }
#endif
    return 0;
}
```

代码中被 HAVE_ROS 宏修饰的部分为 ROS2 适配代码，用户无须关心，不要修改。后续用户添加控制代码时亦无须考虑 ROS 适配。

上述各部分代码，用户仅需在 main 注释处添加控制代码（即 SDK 所提供接口的调用）即可，其余部分不需要修改和维护。

3.4. 信手控制框架解析(SDK-ROS 部分)

SDK-ROS 部分主要功能是将 SDK 电机控制接口封装成 ROS2 的一个节点，电机控制与数据反馈通过 Topic 完成。

Topic 处理代码路径：

sdk2ros \ src \ devices_pkg \ src \ project \ X_Hand_Node \ X_Hand_Node.hpp

结构、变量定义

```
//重复定义结构体
typedef struct X_hand_FB {
    float P;
    float V;
    float F;
    float temp[2];
    u16 error;
} X_hand_FB;
typedef struct X_hand_Send_Data {
    float P;
    float V;
    float F;
    float KP;
    float KD;
} X_hand_Send_Data;
//声明外部变量
extern Robot_Hardware *X_Hand;
extern X_hand_Send_Data Send_Datas[6];
extern X_hand_FB FB_Datas[6];
extern map<u8, vector<u16>> g_sensor_data;
//声明外部函数
int hardware_init(string ADDR);
void Get_FB(void);
void Send(void);
```

重复定义结构是 SDK-ROS 部分与 SDK 部分完全相同的结构，用于声明外部变量时使用与 SDK 相同的结构定义。

外部变量和外部函数的声明部分，是 SDK-ROS 部分涉及到 SDK 部分的变量与处理函数。其中 `Send()` 用于向信手发送控制命令，`Get_FB()` 用于获取信手的反馈数据。

Topic 与 Msg

Topic 定义在 X_Hand_Node 类中：

```
class X_Hand_Node : public rclcpp::Node
{
public:
    X_Hand_Node()
    : Node("x_hand_node")
    {
        hardware_init("src/devices_pkg/src/sdk/config/YAML/X_Hand/
                       out/TOP.yaml");
        publisher_ = this->create_publisher<devices_pkg::msg::XHandMsg>
                     ("x_hand_publisher", 10);
        subscription_ = this->create_subscription<devices_pkg::msg::
                    XHandMsg>("x_hand_subscriber", 10, \
                    std::bind(&X_Hand_Node::topic_callback, this,
        std::placeholders::_1));

         ...
    }
```

- `x_hand_publisher` - 用于向信手发送控制命令
- `x_hand_subscriber` - 用于接收信手的反馈数据

Msg 定义在：*sdk2ros \ src \ devices_pkg \ msg \ XHandMsg.msg*

msg 类似 C 语言结构体，有一层嵌套定义，即 motors 内部还有具体数据，定义在另一个 msg 文件。

```
devices_pkg/Motor[6] motors
devices_pkg/Pressure[6] pressure
```

- `motors` - 电机相关数据，有嵌套定义，类似结构体
- `pressure` - 触觉传感器数据，有嵌套定义，只有一个 `uint16[] finger`

motors 消息内层定义在：sdk2ros \ src \ devices_pkg \ msg \ types \ Motor.msg

```
float32 pos          # 关节电机位置
float32 vel          # 关节电机速度
float32 tor          # 关节电机力矩
float32 kp           # 关节电机 Kp
float32 kd           # 关节电机 Kd
float32[2] temp      # 关节电机温度
uint16 error         #错误
```

可以看到 msg 定义与 X_hand_FB 相同。

Topic 接收处理

Topic 接收回调处理如下：

```
void topic_callback(const devices_pkg::msg::XHandMsg::SharedPtr msg) const
{
    /* 1. 解析收到 msg 中的数据并下发控制命令  */
    // 解析数据
    for (size_t i = 0; i < 6; i++)
    {
        Send_Datas[i].P = msg->motors[i].pos;
        Send_Datas[i].V = msg->motors[i].vel;
        Send_Datas[i].F = msg->motors[i].tor;
        Send_Datas[i].KP = msg->motors[i].kp;
        Send_Datas[i].KD = msg->motors[i].kd;
    }
    // 调用 SDK 接口向信手发送命令
    Send();

    std::this_thread::sleep_for(std::chrono::milliseconds(2));

    /* 2. 立即读取信手的反馈并通过 topic 发布 */
    X_Hand_Node* mutable_this = const_cast<X_Hand_Node*>(this);
    // 调用 SDK 接口读取反馈
    Get_FB();
    // 待全部 6 个触觉传感器数据读取完毕后，再发布
    if(g_sensor_data.size() < 6)
        return;
    // 构建 msg
    for (size_t i = 0; i < 6; i++)
```

```
        {
            mutable_this->sendMes.motors[i].pos = FB_Datas[i].P;
            mutable_this->sendMes.motors[i].vel = FB_Datas[i].V;
            mutable_this->sendMes.motors[i].tor = FB_Datas[i].F;
            mutable_this->sendMes.motors[i].temp[0] = FB_Datas[i].temp[0];
            mutable_this->sendMes.motors[i].temp[1] = FB_Datas[i].temp[1];
            mutable_this->sendMes.motors[i].error = FB_Datas[i].error;

            mutable_this->sendMes.pressure[i].finger = g_sensor_data.
             at((u8)i+1);
        }
        // 发布 msg
        publisher_->publish(sendMes);
    }
```

如上，收到其他节点的 Topic 后作两步处理：先解析 msg 消息数据，然后向信手里发送控制命令；立即获取信手反馈数据并通过 Topic 发布出来。

因为某些运控场景需要实时获取信手当前位置速度等信息，如果采用命令-反馈式交互会拖慢反馈速度，故采用收到控制命令立即反馈的方式完成。

Topic 发布示例

示例代码在：sdk2ros \ src \ devices_pkg \ src \ project \ Test_Node.cpp

```
...
  // 创建 wall timer, 2ms 触发
  timer_ = this->create_wall_timer(
  std::chrono::milliseconds(2),
  std::bind(&Test_Node::timer_callback, this));
...

  // timer 回调
  void timer_callback()
  {
     auto message = devices_pkg::msg::XHandMsg();
     // 每 2s 翻转一次 test 值
     if (times % 1000 == 0) {
         test = -test;
     }
     times++;
```

```
    // 设置电机位置
    for (int i = 0; i < 6; i++) {
        message.motors[i].pos = 0.5 + test;
        message.motors[i].vel = 0;
        message.motors[i].tor = 0;
        message.motors[i].kp = 500;
        message.motors[i].kd = 5;
    }
    // 微调拇指位置，避免碰撞
    message.motors[0].pos = 0.25 + test / 2;
    message.motors[1].pos = 0.1 + test / 5;

    // 发布
    publisher_->publish(message);

}
```

如上示例代码功能是控制信手五指按 2s 周期握拳/张开反复运行。

示例创建了一个 wall timer 用来定时，电机目标位置为 0.5 + test，通过定时翻转 test 正负来实现两个位置轮转切换。

通过 `for` 循环设置所有电机位置后，为避免运动过程中手指间出现碰撞，重新设置了拇指两个电机的目标位置。

这个示例只为说明如何通过 `Topic` 控制电机，更为精细的手部动作控制可通过运动控制算法来实现。

3.5. SDK 控制接口 List

电机使能

函数功能：使能电机。

函数原型：

```
int Motor_EN(shared_ptr<Device_Struct> Device_P, int EN)
```

调用示例：

```
// 使能电机
Motor_1_Control->Motor_EN(Motor_1_D, TRUE);
```

MIT 模式控制电机

函数功能：向电机发送指令，用 MIT 模式控制电机。

函数原型：

```
int Send_MIT_PD_Control_Data(shared_ptr<Device_Struct> Device_P,
                             float Rad,     float Speed_Rad_S,
                             float Force_N, float P_N_Rad,
                             float D_N_Rad_s);
```

参数解析：

- Rad: 目标位置，单位弧度
- Speed_Rad_S: 目标速度，单位弧度/秒
- Force_N: 扭矩前馈，单位 N
- P_N_Rad: KP 参数
- D_N_Rad_S: Kd 参数

调用示例：

```
// MIT 模式控制电机转 100 度
Motor_2_Control->Send_MIT_PD_Control_Data(Motor_2_D, 1.75f, 0, 0, 500, 5);
```

电流模式控制电机

函数功能：纯电流环控制电机。

函数原型：

```
int Send_Current_Control_Data(shared_ptr<Device_Struct> Device_P,
                              float Current_A);
```

参数解析：

- Device_P:　设备指针
- Current_A:　目标电流值，单位安培(A)

调用示例：

```
// 电流模式运行电机,目标电流 0.5A
Motor_3_Control->Send_Current_Control_Data(Motor_3_D, 0.5f);
```

获取电机反馈数据 - 带 error code

函数功能：向电机发送指令，将电机的反馈数据存入相应变量中。

函数原型：

```
int Get_Motor_FB_Data( shared_ptr<Device_Struct> Device_P,
                       float *P, float *V,
                       float *F, float temp[2], u16 *error );
```

参数解析：

- Device_P:　设备指针
- P:　指针，指向当前位置数据
- V:　指针，指向当前速度数据
- F:　指针，指向当前力矩数据
- temp:　保留字段
- error:　错误码

调用示例：

```
// 定义存储反馈数据的变量/数组
float P,V,F;
float temp[2];
u16 error_code;

// 获取反馈数据
Motor_4_Control->Get_Motor_FB_Data(Motor_4_D,
                                   &P, &V, &F, temp, &error_code);
```

获取电机反馈数据 - 不带 error code

函数功能：与上个接口相同，只是不返回错误码，不必填充保留字段。

函数原型：

```
int Get_Motor_FB_Data(shared_ptr<Device_Struct> Device_P,
                      float *P, float *V,  float *F, );
```

参数解析：

- Device_P:　　设备指针
- P:　　指针，指向当前位置数据
- V:　　指针，指向当前速度数据
- F:　　指针，指向当前力矩数据

调用示例：

```
// 定义存储反馈数据的变量/数组
float P,V,F;

// 获取反馈数据
Motor_5_Control->Get_Motor_FB_Data(Motor_5_D, &P, &V, &F);
```

4. 常见问题与解决方案

1. Switch Board 不在线

出现 `ERROR: Main_Switch_Board -> Not_Online !!!` 报错表示信手 Switch Board 板与 SDK 通信失败(如下红色 log 打印):

```
username@hostname:~/.../sdk_2/bin$ ./Test_Motor
[16:45:06][I]#####################################################
[16:45:06][I] Add_Device_Type: Switch_Board
[16:45:06][I] Init_F:    Add Ok
[16:45:06][I] CallBack_F:Add Ok
[16:45:06][I] Delete_F:  Add Ok
[16:45:06][I]#####################################################
[16:45:06][I]#####################################################
[16:45:06][I] Add_Device_Type: Motor_Custom_DaMiao
[16:45:06][I] Init_F:    Add Ok
[16:45:06][I] CallBack_F:Add Ok
[16:45:06][I] Delete_F:  Add Ok
[16:45:06][I]#####################################################

程序所在目录: "/home/username/workspace/sdk_2/bin"
[16:45:06][I] 处理文件: Motor/DaMiao_Motor_Config.yaml
*-*-*-*-*-*-* 完整路径:

[... 中间路径打印省略]

[2026-01-05 16:45:06.340] [LOG] [I] Start_Init_Device

[16:45:06][I]+++++++++++++++++++++++++++++++++++>this:1 all:11
[16:45:06][I] Add_Device_To_Map: Main_Switch_Board
[16:45:06][I] Save_ETH_Device_To_Map:Main_Switch_Board
[16:45:07][E] ERROR: Main_Switch_Board -> Not_Online !!!
[16:45:11][I] Send_RST...
[16:45:14][E] ERROR: Main_Switch_Board -> Not_Online !!!
[16:45:19][I] Send_RST...
[16:45:22][E] ERROR: Main_Switch_Board -> Not_Online !!!
[16:45:22][I] Device->IS_Online_In_Init: 0

[... 后续打印省略]
```

SDK 会重试 3 次，如都失败则打印 `Device->IS_Online_In_Init:0`，表示

Switch Board 初始化失败。

因为 Switch Board 作为 SDK 与电机驱动板间的通信桥梁，所以 Switch Board 初始化失败，会导致 SDK 无法与电机驱动板通信，故无法控制电机。

需要注意的是，如果信手首次上电运行，因为 SDK 需要与主控对接通信 ID，所以也会打印一条 Not_Online 的 log，但重试一次后会成功通信，所以这次错误打印可以忽略。

排查方向:

- 检查 PC 有线连接设置，IP 地址配置是否正确。
  出现 Switch Board 不在线问题，80%概率为有线连接配置问题，优先排查。

  Ubuntu24.04 概率会有锁屏唤醒后有线连接丢失的 bug，重启电脑可解决。

- 检查网线连接、检查信手是否上电
- 检查 Switch Board 固件版本是否与 SDK 版本匹配（出厂时会做匹配测试正常后发货，后期固件升级时需要注意版本匹配问题）