agent_sdk 用户软件开发指导手册

1. 概述

1.1 手册目的

本文档旨在为使用 agent_sdk（SNMP Agent 开发套件）的用户提供完整的开发指导，包括环境搭建、核心功能开发、硬件适配及问题排查，帮助用户快速基于 SDK 实现自定义 SNMP Agent 功能。

> 💡 1. 文档核心定位：指导用户基于 agent_sdk 开源套件，开发适配自身业务场景的 SNMP 软件。
>
> 2. 代码验证说明：文档中提供的各类样例代码（如自定义 OID 实现、传输模块示例等）仅作开发参考，未经过完整功能验证；agent_sdk 开源套件本身包含的开源模块代码，已在 Windows 环境下完成功能验证，可直接使用。

1.2 SDK 简介

agent_sdk 是芯祥联科技推出的基于 xxlsnmp（国产 SNMP 协议栈，替代 netsnmp）的开源开发套件，为用户提供 SNMP Agent 核心框架，支持 SNMP 协议基础交互能力。

> 📌 核心平台支持说明：当前 SDK 提供的 LIB 库仅支持 Windows 环境（x86_64），用户可直接基于该环境开发自定义代码（如自定义 OID、业务逻辑等），并完成基础功能调试；若需移植至 Linux、STM32 等其他硬件平台，需联系芯祥联科技进一步获取合作服务。

用户可基于 SDK 进行二次开发，包括但不限于：添加自定义 OID/MIB、开发专属传输模块、实现个性化数据采集逻辑、定制日志系统。

1.3 核心价值

- **开源可扩展**：核心代码开源，支持用户按需修改和扩展，降低定制化成本。
- **平台支持说明**：当前版本仅完成 Windows 平台调试验证，Linux、STM32 等其他平台需联系芯祥联科技合作开发支持。
- **模块化设计**：按功能拆分公共组件、传输层、硬件适配层等模块，开发聚焦特定场景。
- **闭源库联动**：支持链接专用闭源库，补充加密、高级协议等增强功能。

1.4 适用场景

工业控制、智能硬件、网络设备等需要通过 SNMP 协议实现设备管理、数据上报、远程控制的场景。

2. 开发环境准备

2.1 硬件环境要求

- **仅支持平台**：Windows（x86_64），内存 ⩾ 1GB，存储 ⩾ 100MB；Linux、STM32 等其他硬件平台的适配需求，需联系芯祥联科技获取合作开发支持。
- **自定义硬件**：若基于 Windows 平台扩展硬件，需具备网络通信能力（以太网/Wi-Fi/4G 等，视传输需求而定）。

2.2 软件环境配置

2.2.1 编译环境

目标平台	编译器/工具链	依赖库/工具
Windows	MinGW64（GCC 15.2+）	MSYS2、WSL（可选）
Linux/STM32	暂未开放	需联系芯祥联科技合作适配

2.2.2 开发工具

- **代码编辑**：VS Code、Vim、Keil（STM32 专用）。
- **MIB 开发**：当前 SDK 暂未提供 xxl_mib2c、xxl_mib_check 等配套工具集，需手动编写 MIB 关联的 C 语言代码框架。
- **调试工具**：Wireshark（抓包）、xxlsnmp 自研命令行工具（xxlsnmpnms，国产替代方案，随SDK开源给用户使用）、GDB（程序调试）。
- **版本控制**：Git（管理自定义代码）。

2.2.3 SDK 获取与目录结构

1. 获取 SDK 源码，放置于开发目录（以 Linux 为例，路径为 `/home/user/project/agent_sdk`）。

2. 核心目录结构及说明：

- `./comm/`：公共组件（日志、内存管理、xxlsnmp 协议解析等核心逻辑）。
- `./platform/`：硬件适配层（不同平台的延时、网络、外设接口）。
- `./tps/`：目标协议栈数据采集模块（针对特定网络协议栈如 AFDX，调用对应网络接口完成 OID 数据采集，基于 xxlsnmp 框架封装）。

- `./trans/`：传输层（UDP/TCP 通信，支持自定义传输）。
- `./usermib/`：用户 MIB 模板（自定义 OID 开发入口）。
- `./inc/`：公共头文件（SDK 对外接口、宏定义）。

> 💡 重要提示：./inc/目录下SDK对外提供的头文件（如trans.h、platform.h、xxl_snmp.h等）包含核心接口定义和宏配置，**禁止修改其内容**。擅自修改可能导致头文件与SDK内部逻辑不兼容，引发编译失败、Agent程序崩溃或功能异常等问题。若需扩展接口，可基于现有接口进行二次封装，而非直接修改原头文件。

- `./lib/`：闭源库目录（按平台分类，如 linux/、stm32/）。
- `./bin/`：编译输出目录（自动生成）。
- `./build_agent.sh`：编译脚本（核心配置与编译入口）。

3. 基础开发流程

3.1 编译脚本配置

编译脚本 `build_agent.sh` 是 SDK 编译的核心，用户需根据目标平台和需求调整关键配置：

3.1.1 核心配置项

- **目标平台**：当前 SDK LIB 仅支持 Windows 平台，必须通过 `-p windows` 参数指定；Linux、STM32 等其他平台的支持需联系芯祥联科技合作开发，暂不提供独立适配能力。
- **源码目录**：`OPEN_SRC_DIRS` 定义需要编译的开源目录，新增自定义目录需添加至此。
- **头文件目录**：`INC_DIRS` 定义编译依赖的头文件路径，自定义头文件需补充。
- **宏定义**：在各平台 `CFLAGS` 中添加宏，控制 SDK 功能开关。
- **输出配置**：`OUTPUT_ROOT` 定义编译产物路径，`TARGET_BASE_NAME` 定义目标文件前缀。

3.1.2 编译执行示例

Code block

```
# 1. 给脚本添加执行权限（MinGW 环境）
chmod +x build_agent.sh

# 2. 编译Windows平台（当前仅支持此平台）
./build_agent.sh -p windows

# 3. 编译完成后，产物位于 ./bin/windows/ 目录下
# 产物：xxl_agent_snmp_demo_windows.exe
```

3.2 基础功能验证

编译完成后，需验证 SDK 基础功能是否正常，确保开发环境无误：

3.2.1 通用验证（Linux/Windows）

Code block

```
# 1. 启动编译生成的Agent程序（Windows MinGW/CMD环境）
./bin/windows/xxl_agent_snmp_demo_windows.exe

# 2. 另开终端，使用xxlsnmp工具测试（自研xxlsnmpnms，随SDK开源）
# 测试系统默认OID（1.3.6.1.2.1.1为系统信息，getnext实现遍历功能）
xxlsnmpnms getnext -v 2c -c public 127.0.0.1 1.3.6.1.2.1.1
```

若能正常返回系统信息，说明 SDK 基础通信功能正常。

3.2.2 嵌入式验证（STM32）

注：当前版本暂不支持 STM32 嵌入式平台，该平台适配需与芯祥联科技进一步合作。

4. 核心功能开发

4.1 自定义 OID/MIB 开发

MIB（管理信息库）定义了SNMP可管理的对象（OID对应唯一对象）。当前SDK暂未提供xxlsnmp的MIB文件开发工具，不支持通过MIB文件形式导入OID，需通过“定义OID常量→实现业务回调→注册到SDK”的手动编码方式完成OID导入与开发，全程基于国产xxlsnmp框架，替代传统netsnmp方案。

4.1.1 核心说明：OID导入的本质

💡 由于缺乏MIB解析工具，“导入OID”在当前SDK中表现为：手动将OID对应的常量、数据处理逻辑编码实现，并通过SDK提供的 `xxl_oid_register` 接口注册到Agent框架中，完成OID与业务逻辑的绑定。

4.1.2 手动导入OID：完整实现步骤

以导入“设备温度（1.3.6.1.4.1.12345.2）”和“设备状态（1.3.6.1.4.1.12345.1）”两个OID为例，具体步骤如下：

步骤1：定义OID常量与接口（头文件）

在 `./usermib/userDevice.h` 中定义OID对应的常量（模拟MIB文件的OID映射）及回调函数声明，实现OID“导入”的基础定义：

Code block

```
#ifndef USER_DEVICE_H
#define USER_DEVICE_H

#include "xxl_snmp.h"  // 引入SDK核心头文件

// 1. 定义OID常量（对应1.3.6.1.4.1.12345， enterprises=1.3.6.1.4.1，用户企业ID=12345）
#define OID_ENTERPRISES       1,3,6,1,4,1
#define OID_USER_DEVICE_BASE  OID_ENTERPRISES,12345
// 设备状态OID：1.3.6.1.4.1.12345.1
#define OID_DEVICE_STATUS     OID_USER_DEVICE_BASE,1
// 设备温度OID：1.3.6.1.4.1.12345.2
#define OID_DEVICE_TEMP       OID_USER_DEVICE_BASE,2

// 2. 声明OID对应的数据处理回调函数（SNMP请求触发时执行）
int get_deviceStatus(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
                     xxl_snmp_request *reqinfo, xxl_snmp_data *requests);
int get_deviceTemp(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
                   xxl_snmp_request *reqinfo, xxl_snmp_data *requests);

// 3. 声明OID注册（导入）入口函数
void user_oid_init();

#endif
```

步骤2：实现OID业务逻辑（源文件）

在 `./usermib/userDevice.c` 中实现回调函数，完成OID与实际业务数据的关联：

Code block

```
#include "userDevice.h"
#include "platform.h"  // 硬件适配层，用于获取实际设备数据

// 设备状态读取回调（OID：1.3.6.1.4.1.12345.1）
int get_deviceStatus(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
                     xxl_snmp_request *reqinfo, xxl_snmp_data *requests) {
    // 实际场景：调用硬件接口获取设备状态（0=故障，1=正常）
    int status = platform_check_status();  // 硬件适配层接口，需用户实现

    // 将数据赋值给SNMP响应（XXL_ASN_INTEGER对应数据类型）
```

```
    xxl_snmp_set_value(requests, XXL_ASN_INTEGER, (u_char*)&status,
sizeof(status));
    return XXL_SNMP_ERR_NOERROR;  // 返回无错误，完成响应
}

// 设备温度读取回调（OID：1.3.6.1.4.1.12345.2）
int get_deviceTemp(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
                   xxl_snmp_request *reqinfo, xxl_snmp_data *requests) {
    // 实际场景：调用温度传感器接口获取数据（单位：℃）
    int temp = platform_read_temp();  // 硬件适配层接口，需用户实现

    xxl_snmp_set_value(requests, XXL_ASN_INTEGER, (u_char*)&temp,
sizeof(temp));
    return XXL_SNMP_ERR_NOERROR;
}
```

步骤3：注册OID到SDK（完成导入）

在 `./usermib/userDevice.c` 中实现 `user_oid_init` 函数，通过SDK接口将OID“导入”到Agent框架：

Code block

```
// OID注册（导入）入口：将自定义OID绑定到SDK
void user_oid_init() {
    // 1. 注册设备状态OID
    xxl_oid_register(
        "deviceStatus",          // OID名称（仅用于调试标识）
        get_deviceStatus,        // 绑定回调函数
        OID_DEVICE_STATUS,       // 待导入的OID常量
        XXL_OID_READ_ONLY        // 权限（只读/读写）
    );

    // 2. 注册设备温度OID
    xxl_oid_register(
        "deviceTemp",
        get_deviceTemp,
        OID_DEVICE_TEMP,
        XXL_OID_READ_ONLY
    );

    agent_log("INFO", __FILE__, __LINE__, "自定义OID导入完成：设备状态、设备温度");
}
```

步骤4：配置SDK加载OID（初始化入口）

在 `./usermib/user_mib.c` 的初始化函数中调用 `user_oid_init`，确保Agent启动时自动导入OID：

Code block

```
#include "userDevice.h"
#include "xxl_snmp.h"

// SDK默认调用的MIB初始化函数
void user_mib_init() {
    // 触发自定义OID的导入与注册
    user_oid_init();
    // 其他MIB初始化逻辑...
}
```

在 `./usermib/` 目录下新建 MIB 文件（如 `USER-DEVICE-MIB.mib`），示例如下（定义设备温度和设备状态两个 OID）：

Code block

```
USER-DEVICE-MIB DEFINITIONS ::= BEGIN
    IMPORTS
        enterprises FROM SNMPv2-SMI
        Integer32 FROM SNMPv2-SMI;

    userDevice MODULE-IDENTITY
        LAST-UPDATED "202512160000Z"
        ORGANIZATION "用户企业名称"
        CONTACT-INFO "技术支持：xxx@xxx.com"
        DESCRIPTION "用户设备自定义MIB"
        ::= { enterprises 12345 }  -- 12345为用户分配的企业OID（可向IANA申请）

    deviceStatus OBJECT-TYPE
        SYNTAX      Integer32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION "设备运行状态：0=故障，1=正常"
        ::= { userDevice 1 }  -- 完整OID：1.3.6.1.4.1.12345.1

    deviceTemp OBJECT-TYPE
        SYNTAX      Integer32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION "设备温度（单位：℃）"
        ::= { userDevice 2 }  -- 完整OID：1.3.6.1.4.1.12345.2
END
```

4.1.2 手动创建代码框架

当前 SDK 暂未提供 xxl_mib2c 工具，需基于 MIB 文件定义的 OID 信息，手动创建 C 语言代码框架（替代工具生成流程），具体步骤如下：

Code block

```
# 1. 在 ./usermib/ 目录下新建 userDevice.h 头文件，定义接口声明
# 2. 新建 userDevice.c 源文件，实现 OID 数据处理逻辑
# 无需依赖工具链，直接基于 SDK 接口编写即可
```

使用 xxlsnmp 配套工具 `xxl_mib2c`（随 SDK 安装包提供，国产替代 mib2c）将 MIB 文件转换为 C 语言代码框架，无需依赖 netsnmp 工具链：

Code block

```
# 1. 导出MIB路径（让xxlsnmp工具识别自定义MIB）
export XXL_MIBDIRS=./usermib/:$XXL_MIBDIRS

# 2. 用xxlsnmp工具生成代码（xxl_mib2c为国产替代工具）
xxl_mib2c -c xxl_scalar.conf userDevice
```

手动创建的代码文件需包含 OID 注册所需的回调函数声明和实现，核心结构与 xxlsnmp 框架接口适配，与 netsnmp 接口完全隔离。

4.1.3 实现业务逻辑

修改生成的 `userDevice.c`，在数据读取函数中补充自定义采集逻辑：

Code block

```
#include "userDevice.h"
#include "platform.h"  // 硬件适配层接口，用于读取温度
#include "xxl_snmp.h"  // xxlsnmp核心头文件，替代netsnmp相关头文件

// 设备状态读取函数（xxlsnmp回调接口，替代netsnmp_handler）
int get_deviceStatus(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
                     xxl_snmp_request *reqinfo, xxl_snmp_data *requests) {
    int status = 1;  // 模拟设备正常状态，实际需结合硬件状态判断

    // 给SNMP请求赋值（xxlsnmp接口，替代snmp_set_var_typed_value）
    xxl_snmp_set_value(requests, XXL_ASN_INTEGER, (u_char*)&status,
sizeof(status));
    return XXL_SNMP_ERR_NOERROR;
}
```

```
14
15  // 设备温度读取函数（xxlsnmp回调接口）
16  int get_deviceTemp(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
17                     xxl_snmp_request *reqinfo, xxl_snmp_data *requests) {
18      int temp = platform_read_temp();  // 调用硬件适配层接口读取实际温度
19
20      xxl_snmp_set_value(requests, XXL_ASN_INTEGER, (u_char*)&temp,
    sizeof(temp));
21      return XXL_SNMP_ERR_NOERROR;
22  }
```

4.1.4 注册 OID 到 SDK

在 `./usermib/user_mib.c` 的 `user_mib_init()` 函数中，添加自定义 OID 注册逻辑：

Code block

```
1   #include "userDevice.h"
2   #include "xxl_snmp.h"  // xxlsnmp注册接口头文件
3
4   void user_mib_init() {
5       // 注册设备状态OID（xxl_oid_register替代netsnmp_register_scalar）
6       xxl_oid_register(
7           "deviceStatus",          // OID名称
8           get_deviceStatus,        // 回调函数
9           XXL_OID_USER_DEVICE_1,   // OID常量（由xxl_mib2c生成）
10          XXL_OID_READ_ONLY        // 只读权限
11      );
12
13      // 注册设备温度OID（xxlsnmp注册接口）
14      xxl_oid_register(
15          "deviceTemp",
16          get_deviceTemp,
17          XXL_OID_USER_DEVICE_2,
18          XXL_OID_READ_ONLY
19      );
20  }
```

4.1.5 编译与测试

步骤5：编译配置与OID导入验证

1. **编译配置**：确保 `userDevice.c` 被纳入编译（`build_agent.sh` 的 `OPEN_SRC_DIRS` 已包含 `./usermib`，无需额外修改）。

2. **编译执行**：

Code block

```
# Windows MinGW环境编译
chmod +x build_agent.sh
./build_agent.sh -p windows

# 启动Agent（OID会随Agent启动自动导入）
./bin/windows/xxl_agent_snmp_demo_windows.exe
```

3. **验证OID导入成功**：使用自研工具 `xxlsnmpnms` 读取OID，确认导入生效：

Code block

```
# 读取设备温度（验证OID 1.3.6.1.4.1.12345.2导入成功）
xxlsnmpnms get -v 2c -c public 127.0.0.1 1.3.6.1.4.1.12345.2

# 读取设备状态（验证OID 1.3.6.1.4.1.12345.1导入成功）
xxlsnmpnms get -v 2c -c public 127.0.0.1 1.3.6.1.4.1.12345.1

# 批量遍历已导入的自定义OID树
xxlsnmpnms bulkwalk -v 2c -c public 127.0.0.1 1.3.6.1.4.1.12345
```

4.1.3 OID导入常见问题

问题现象	原因分析	解决方法
读取OID返回“noSuchObject”	1. OID常量定义错误；2. `user_oid_init` 未被调用；3. 注册接口参数错误	1. 核对OID常量与实际读取的OID是否一致；2. 确认 `user_mib_init` 中已调用 `user_oid_init` ；3. 检查 `xxl_oid_register` 的OID参数是否完整
Agent启动报“undefined reference to user_oid_init”	`userDevice.c` 未被编译进目标程序	检查 `build_agent.sh` 的 `OPEN_SRC_DIRS` ，确保 `./usermib` 在目录列表中

2. 编译后启动 Agent，执行测试命令：

Code block

```
# 读取设备温度（替换[AgentIP]为实际IP，使用自研xxlsnmpnms工具）
xxlsnmpnms get -v 2c -c public [AgentIP] 1.3.6.1.4.1.12345.2

```

```
# 读取设备状态（xxlsnmpnms工具，随SDK开源）
xxlsnmpnms get -v 2c -c public [AgentIP] 1.3.6.1.4.1.12345.1
```

4.2 自定义传输模块开发

SDK 默认支持 UDP 传输，用户可在 `./trans/` 目录下开发自定义传输（如 TCP、串口透传），核心是实现传输层接口规范。

4.2.1 传输层接口定义

在 `./inc/trans.h` 中定义传输层统一接口，用户需实现以下函数：

Code block

```
// 传输模块初始化（如创建socket、配置串口）
int trans_init(const char *addr, int port);

// 发送SNMP消息
int trans_send(const u_char *data, int len, const char *dst_addr, int
dst_port);

// 接收SNMP消息（阻塞/非阻塞，按需实现）
int trans_recv(u_char *buf, int buf_len, char *src_addr, int *src_port);

// 传输模块销毁（释放资源）
void trans_destroy();
```

4.2.2 实现 TCP 传输示例

在 `./trans/` 目录下新建 `trans_tcp.c`，实现 TCP 传输逻辑：

Code block

```
#include "trans.h"
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

static int sock_fd = -1;
static struct sockaddr_in server_addr;

// 初始化TCP服务端（Agent作为服务端监听）
int trans_init(const char *addr, int port) {
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd < 0) return -1;

```

```
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(addr);
    server_addr.sin_port = htons(port);

    // 绑定端口
    if (bind(sock_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        close(sock_fd);
        return -1;
    }

    // 监听连接
    listen(sock_fd, 5);
    return 0;
}

// 接收消息（需先accept连接，简化示例）
int trans_recv(u_char *buf, int buf_len, char *src_addr, int *src_port) {
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);
    int conn_fd = accept(sock_fd, (struct sockaddr*)&client_addr, &client_len);
    if (conn_fd < 0) return -1;

    int recv_len = recv(conn_fd, buf, buf_len, 0);
    strcpy(src_addr, inet_ntoa(client_addr.sin_addr));
    *src_port = ntohs(client_addr.sin_port);

    close(conn_fd);
    return recv_len;
}

// 发送消息（需指定TCP客户端地址）
int trans_send(const u_char *data, int len, const char *dst_addr, int dst_port) {
    struct sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_addr.s_addr = inet_addr(dst_addr);
    client_addr.sin_port = htons(dst_port);

    int conn_fd = connect(sock_fd, (struct sockaddr*)&client_addr, sizeof(client_addr));
    if (conn_fd < 0) return -1;

    int send_len = send(conn_fd, data, len, 0);
    close(conn_fd);
    return send_len;
}
```

```
58
59  void trans_destroy() {
60      if (sock_fd >= 0) close(sock_fd);
61  }
```

4.2.3 切换传输模块

在 `./tps/snmp_agent.c` 中，将默认 UDP 传输切换为自定义 TCP 传输：

Code block

```
1   #include "trans.h"
2   #include "xxl_snmp.h"  // xxlsnmp初始化依赖
3
4   void agent_init() {
5       // 初始化xxlsnmp核心框架
6       xxl_snmp_core_init();
7       // 替换为TCP传输初始化，监听161端口（SNMP默认端口）
8       trans_init("0.0.0.0", 161);
9       // 初始化用户自定义MIB
10      user_mib_init();
11      // ... 其他初始化逻辑
12  }
```

4.3 自定义 OID 数据采集

OID 数据采集逻辑需与硬件状态或业务数据关联，用户可通过“硬件适配层接口 → 数据处理 → OID 回调”实现，支持实时采集和缓存采集两种模式。

4.3.1 实时采集模式

适用于数据实时性要求高的场景，每次 SNMP 请求触发时直接读取硬件数据（如 4.1.3 中的温度采集），核心是调用硬件适配层接口。

4.3.2 缓存采集模式

适用于数据更新频率低、采集耗时的场景，通过定时器周期性采集并缓存数据，OID 回调直接读取缓存：

Code block

```
1   #include "userDevice.h"
2   #include "platform.h"
3   #include "xxl_snmp.h"
4
5   // 缓存变量
```

```
static int g_device_temp = 0;
static int g_device_status = 1;

// 周期性采集线程（基于xxlsnmp定时器接口实现，替代自定义线程）
void data_collect_thread() {
    // 注册xxlsnmp定时器，1秒执行一次
    xxl_snmp_timer_register(1000, 1, collect_temp_status);
}

// 定时器回调：采集温度和状态
void collect_temp_status() {
    g_device_temp = platform_read_temp();  // 读取硬件温度
    g_device_status = platform_check_status();  // 检查设备状态
}

// OID回调函数读取缓存（xxlsnmp适配接口）
int get_deviceTemp(xxl_mib_handler *handler, xxl_oid_reg_info *reginfo,
                   xxl_snmp_request *reqinfo, xxl_snmp_data *requests) {
    xxl_snmp_set_value(requests, XXL_ASN_INTEGER,
                           (u_char*)&g_device_temp, sizeof(g_device_temp));
    return XXL_SNMP_ERR_NOERROR;
}
```

4.4 自定义日志记录

SDK 默认日志输出到控制台，用户可在 `./comm/agent_log.c` 中修改日志模块，支持输出到文件、串口或远程日志服务器。

4.4.1 日志输出到文件示例

Code block

```
#include "agent_log.h"
#include <stdio.h>
#include <time.h>

static FILE *log_fd = NULL;

// 日志初始化（打开日志文件）
void log_init(const char *log_path) {
    log_fd = fopen(log_path, "a+");
    if (!log_fd) {
        // 初始化失败时退化为控制台输出
        log_fd = stdout;
    }
}
```

```
15
16  // 自定义日志输出函数
17  void agent_log(const char *level, const char *file, int line, const char *fmt,
    ...) {
18      va_list args;
19      va_start(args, fmt);
20
21      // 添加时间戳
22      time_t now = time(NULL);
23      char time_str[32];
24      strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", localtime(&now));
25
26      // 输出格式: 时间 级别 文件名:行号 日志内容
27      fprintf(log_fd, "[%s] [%s] %s:%d ", time_str, level, file, line);
28      vfprintf(log_fd, fmt, args);
29      fprintf(log_fd, "\n");
30      fflush(log_fd);  // 强制刷新，避免日志丢失
31
32      va_end(args);
33  }
34
35  // 日志模块销毁
36  void log_destroy() {
37      if (log_fd && log_fd != stdout) {
38          fclose(log_fd);
39      }
40  }
```

4.4.2 日志模块调用

在 Agent 初始化函数中初始化日志：

Code block

```
1   #include "xxl_snmp.h"
2   #include "agent_log.h"
3
4   void agent_init() {
5       // 初始化xxlsnmp核心
6       xxl_snmp_core_init();
7       // 初始化日志，输出到./agent.log
8       log_init("./agent.log");
9       agent_log("INFO", __FILE__, __LINE__, "Agent init success");
10      // ... 其他初始化
11  }
```

5. 硬件平台适配开发

SDK 通过 `./platform/` 目录实现硬件解耦，当前该目录下仅提供 Windows 平台基础适配接口。用户可基于 Windows 环境开发自定义硬件的适配代码并完成调试；若需适配 Linux、STM32 等其他硬件平台，需联系芯祥联科技进一步获取合作服务。核心适配接口包括延时、网络、外设操作三类。

5.1 适配层接口规范

在 `./inc/platform.h` 中定义适配层接口，用户需在 `./platform/[硬件名]/` 目录下实现（如 `./platform/custom_hw/`）：

Code block

```
#ifndef PLATFORM_H
#define PLATFORM_H

// 1. 延时接口（毫秒级）
void platform_delay_ms(int ms);

// 2. 网络初始化（返回IP地址字符串）
char* platform_net_init();

// 3. 硬件状态检查（返回0=故障，1=正常）
int platform_check_status();

// 4. 温度读取（返回温度值，单位℃）
int platform_read_temp();

// 5. 外设控制（如LED、继电器，示例接口）
void platform_periph_ctrl(int periph_id, int status);

#endif
```

5.2 自定义硬件适配实现

以“基于ARM Linux的自定义硬件”为例，在 `./platform/custom_hw/platform.c` 中实现接口：

Code block

```
#include "platform.h"
#include <unistd.h>
#include <stdio.h>
#include <string.h>

// 延时接口（调用系统usleep）
```

```c
void platform_delay_ms(int ms) {
    usleep(ms * 1000);
}

// 网络初始化（读取系统IP，简化示例）
char* platform_net_init() {
    static char ip[16] = "192.168.1.100";
    // 实际可通过ioctl获取网卡IP
    return ip;
}

// 硬件状态检查（读取GPIO电平，示例）
int platform_check_status() {
    FILE *fp = fopen("/sys/class/gpio/gpio1/value", "r");
    if (!fp) return 0;
    char val = fgetc(fp);
    fclose(fp);
    return (val == '1') ? 1 : 0;
}

// 温度读取（读取传感器文件，示例）
int platform_read_temp() {
    FILE *fp = fopen("/sys/bus/i2c/devices/1-0048/temp1_input", "r");
    if (!fp) return -1;
    int temp;
    fscanf(fp, "%d", &temp);
    fclose(fp);
    return temp / 1000;  // 转换为℃
}
```

5.3 适配层切换与编译

1. 将自定义硬件目录添加到编译脚本的 `OPEN_SRC_DIRS` 和 `INC_DIRS`（仅Windows平台适配场景）：
```
OPEN_SRC_DIRS=(
    "./comm"
    "./platform/custom_hw"  # 新增自定义硬件目录（Windows平台）
    "./tps"
    "./trans"
    "./usermib"
)
INC_DIRS=(
    "./inc"
    "./platform/custom_hw"  # 新增头文件目录
    # ... 其他目录
)
```
1. 编译对应平台，SDK 会自动链接自定义适配层代码。

6. 进阶开发与优化

6.1 多平台适配技巧

- **条件编译**：当前仅支持Windows平台，可使用平台宏 `WIN32` 实现兼容性控制，后续扩展平台时可补充：
```
    void platform_delay_ms(int ms) {
#ifdef WIN32
    Sleep(ms);  // Windows延时接口
#else
    // 其他平台需联系芯祥联科技合作实现
    #error "当前仅支持Windows平台，其他平台请联系技术支持"
#endif
}
```
- **库文件管理**：将不同平台的闭源库按目录分类（如 `./lib/custom_hw/`），在编译脚本中按平台匹配链接。

6.2 性能优化建议

- **代码裁剪**：基于Windows平台需求，在编译脚本中移除非必要模块（如AFDX协议无关代码、其他平台适配代码），仅保留xxlsnmp核心和Windows相关逻辑。
- **内存优化**：嵌入式平台使用 `-Os` 编译选项（优化体积），避免频繁动态内存分配，使用静态缓冲区。
- **并发优化**：多线程场景下，为共享数据添加互斥锁，避免 SNMP 消息处理冲突。

6.3 安全增强

- **访问控制**：在 OID 注册时限制访问权限（如 `HANDLER_CAN_RW` 仅开放给信任主机）。
- **输入校验**：对 SNMP 请求中的 OID 和数据长度进行校验，防止恶意请求攻击。

7. 问题排查与调试

7.1 常见编译错误

错误现象	可能原因	解决方法
undefined reference to 'WinMain'	Windows 平台编译时，链接器入口错误	在 LDFLAGS 中添加 `-Wl,-e,_mainCRTStartup`
implicit declaration of function '__NOP'	ARM 专属函数在 x86 平台编译	添加宏 `-D__NOP\(\)=\(\)` 替换为空操作
未找到闭源库文件	库文件路径错误或未放置到指定目录	检查 `SDK_LIB_FILE` 配置，将库文件放入 `./lib/[平台名]/`

7.2 运行时问题调试

- **核心测试命令（自定义OID验证）**：

```
# 1. 基础连通性（验证Agent存活，使用系统默认OID）
xxlsnmpnms get -v 2c -c public 127.0.0.1 1.3.6.1.2.1.1.1.0

# 2. 自定义OID读取（替换为实际自定义OID，如设备温度1.3.6.1.4.1.12345.2）
xxlsnmpnms get -v 2c -c public [AgentIP] 1.3.6.1.4.1.12345.2

# 3. 批量遍历自定义OID树（高效验证多个OID）
xxlsnmpnms bulkwalk -v 2c -c public -Cn0 -Cr5 [AgentIP] 1.3.6.1.4.1.12345

# 4. 可写OID测试（若自定义OID支持SET，替换对应OID和值）
xxlsnmpnms set -v 2c -c private [AgentIP] 1.3.6.1.4.1.12345.3 i 1
# i表示整数类型
```

- **SNMP 通信失败**：
  用 ping 命令验证 Agent 与测试主机网络互通：`ping [AgentIP]`
- 检查161端口占用：`netstat -ano -p udp | findstr :161`

- 临时关闭防火墙测试：`netsh advfirewall set allprofiles state off`（测试后需重新开启）

7.3 调试工具使用

- **SNMP 命令行工具**：xxlsnmp自研核心工具xxlsnmpnms（支持get/getnext/set等功能，随SDK开源给用户，替代传统snmpget/snmpwalk）。
- **SNMP 命令行工具**：xxlsnmp配套工具（`xxl_snmpget` 读取OID、`xxl_snmpwalk` 遍历OID树、`xxl_snmpset` 写入OID数据）。
- **日志调试**：开启 SDK 调试日志（编译脚本添加 `-DDEBUG` 宏），输出详细流程信息。

7.4 Net-SNMP与自研Agent对接测试命令汇总（Windows 11环境）

7.4.1 环境配置与验证命令

以下命令均在PowerShell环境执行，涉及权限操作需以管理员身份运行。

Code block

```
# 1. 验证xxlsnmp工具安装成功（替代Net-SNMP的snmp工具）
xxlsnmpnms -h

# 2. 配置Windows防火墙（允许SNMP通信端口161/UDP）
New-NetFirewallRule -DisplayName "SNMP Agent (161/UDP)" -Direction Inbound -LocalPort 161 -Protocol UDP -Action Allow
New-NetFirewallRule -DisplayName "SNMP Agent (161/UDP)" -Direction Outbound -LocalPort 161 -Protocol UDP -Action Allow

# 3. 检查161端口占用情况（解决端口冲突问题）
netstat -ano -p udp | findstr :161

# 4. 结束占用161端口的进程（替换<PID号>为实际进程ID）
taskkill /F /PID <PID号>

# 5. SNMPv3用户配置（编辑C:\usr\etc\snmp\snmpd.conf添加以下内容）
# createUser testuser SHA "AuthPass123" AES "PrivPass123"
# rwuser testuser authPriv
```

7.4.2 SNMPv1测试命令

Code block

```
# 1. 基础连通性测试（读取系统信息OID）
xxlsnmpnms get -v 1 -c public 127.0.0.1 1.3.6.1.2.1.1.1.0  # 系统描述
xxlsnmpnms get -v 1 -c public 127.0.0.1 1.3.6.1.2.1.1.3.0  # 系统运行时间

# 2. GETNEXT操作测试（遍历MIB树，支持输出到日志）
xxlsnmpnms getnext -v 1 -c public 127.0.0.1 1.3.6.1.2.1.1  # 遍历系统信息MIB
xxlsnmpnms getnext -v 1 -c public 192.168.3.134:161 1.3.6.1.4.1.12345  # 遍历自定义MIB
xxlsnmpnms getnext -v 1 -c public 127.0.0.1 1.3.6.1.2.1.1 > snmpwalk_v1.log  # 结果保存到日志

# 3. SET操作测试（修改可写OID，需匹配对应共同体名）
# 读取当前值
xxlsnmpnms get -v 1 -c private 127.0.0.1 1.3.6.1.2.1.1.4.0
# 执行修改（s表示字符串类型）
xxlsnmpnms set -v 1 -c private 127.0.0.1 1.3.6.1.2.1.1.4.0 s "test@windows.com"
# 验证修改结果
xxlsnmpnms get -v 1 -c private 127.0.0.1 1.3.6.1.2.1.1.4.0

# 4. 异常场景测试（权限错误、OID不存在、权限不足）
xxlsnmpnms get -v 1 -c wrongcommunity 127.0.0.1 1.3.6.1.2.1.1.1.0  # 错误共同体名
xxlsnmpnms get -v 1 -c public 127.0.0.1 1.3.6.1.2.1.999.999.0      # 不存在的OID
xxlsnmpnms set -v 1 -c public 127.0.0.1 1.3.6.1.2.1.1.1.0 s "invalid"  # 只读OID写操作
```

7.4.3 SNMPv2c测试命令

Code block

```
# 1. 基础功能验证（批量查询、MIB遍历）
xxlsnmpnms get -v 2c -c public 127.0.0.1 1.3.6.1.2.1.1.1.0 1.3.6.1.2.1.1.3.0  # 批量读取
xxlsnmpnms getnext -v 2c -c public 127.0.0.1 1.3.6.1.2.1.1  # 遍历系统信息MIB
xxlsnmpnms getnext -v 2c -c public 127.0.0.1 1.3.6.1.4.1.12345.1  # 遍历自定义OID

# 2. GETBULK操作测试（v2c特有批量获取，-Cn0起始索引，-Cr5获取数量）
xxlsnmpnms bulkwalk -v 2c -c public -Cn0 -Cr5 127.0.0.1 1.3.6.1.2.1.1

# 3. 错误处理测试（数据类型不匹配）
xxlsnmpnms set -v 2c -c public 127.0.0.1 1.3.6.1.2.1.1.3.0 s "not-integer"  # 时间戳OID写字符串
```

7.4.4 调试与日志命令

Code block

```
# 1. 显示十六进制报文（协议层调试）
xxlsnmpnms get -v 2c -c public -d 127.0.0.1 1.3.6.1.2.1.1.1.0

# 2. 输出全流程详细调试信息（-D all开启所有调试日志）
xxlsnmpnms get -v 2c -c public -D all 127.0.0.1 1.3.6.1.2.1.1.1.0

# 3. 调试日志保存到文件（2>&1将错误信息一并保存）
xxlsnmpnms get -v 2c -c public -D all 127.0.0.1 1.3.6.1.2.1.1.1.0 >
C:\snmp_debug.log 2>&1
```

Code block

```
# 1. 显示十六进制报文（协议层调试）
xxlsnmpnms get -v 2c -c public -d 127.0.0.1 1.3.6.1.2.1.1.1.0

# 2. 输出全流程详细调试信息（-D all开启所有调试日志）
xxlsnmpnms get -v 2c -c public -D all 127.0.0.1 1.3.6.1.2.1.1.1.0

# 3. 调试日志保存到文件（2>&1将错误信息一并保存）
xxlsnmpnms get -v 2c -c public -D all 127.0.0.1 1.3.6.1.2.1.1.1.0 >
C:\snmp_debug.log 2>&1
```

7.4.5 TRAP/INFORM触发与监听测试

触发逻辑：SET操作控制LED状态（OID：1.3.6.1.4.1.12345.1.2），LED=0触发链路断开Trap，LED=1触发链路恢复Inform。

Code block

```
# 步骤1：环境准备（管理员权限）
# 开放TRAP监听端口162/UDP
New-NetFirewallRule -DisplayName "SNMP TRAP (162/UDP)" -Direction Inbound -
LocalPort 162 -Protocol UDP -Action Allow
# 检查161/162端口占用
netstat -ano -p udp | findstr :161
netstat -ano -p udp | findstr :162
# 启动自研Agent（确保加载user_mib模块，监听127.0.0.1:161）

# 步骤2：启动TRAP监听（新终端窗口，保持运行）
xxlsnmpnms traplisten -p 162

# 步骤3：触发TRAP/INFORM（另一个终端窗口）
# 读取当前LED状态
xxlsnmpnms get -v 2c -c private 127.0.0.1 1.3.6.1.4.1.12345.1.2
# SET LED=1（开启）→ 触发链路恢复Inform
xxlsnmpnms set -v 2c -c private 127.0.0.1 1.3.6.1.4.1.12345.1.2 i 1
```

```
# SET LED=0（关闭）→ 触发链路断开Trap
xxlsnmpnms set -v 2c -c private 127.0.0.1 1.3.6.1.4.1.12345.1.2 i 0

# 步骤4：验证监听结果
# 监听窗口应输出类似日志（Inform示例）：
# ========================================
# [TRAP/INFORM] 来源：127.0.0.1:161（版本：v2c）
# ========================================
# 安全信息：共同体名 = public
# 变量数据（共2个）：
#   [1] OID：1.3.6.1.2.1.1.3.0 = TIMETICKS：45678（456.78秒）
#   [2] OID：1.3.6.1.6.3.1.1.4.1.0 = OID：1.3.6.1.6.3.1.1.5.2（链路恢复陷阱OID）
# ----------------------------------------
```

触发逻辑：SET操作控制LED状态（OID：1.3.6.1.4.1.12345.1.2），LED=0触发链路断开Trap，LED=1触发链路恢复Inform。

8.1 常用参考资料

- SNMP 协议规范：RFC 1157（SNMPv1）、RFC 1901（SNMPv2c）、RFC 3411（SNMPv3）。
- xxlsnmp 开发文档：芯祥联科技官网（官网内可获取完整开发文档及资源）。
- MinGW 工具链使用指南：MinGW-w64 官方文档。

8.2 编译脚本参数说明

参数	功能	示例
-p <platform>	指定目标平台	./build_agent.sh -p stm32
-h/--help	查看帮助信息	./build_agent.sh -h

8.3 版本说明

本手册对应 agent_sdk v1.0 版本，若 SDK 版本更新，可参考配套的版本变更日志调整开发流程。

8.4 技术支持

若遇到开发问题，或需要Linux、STM32等其他硬件平台的移植适配支持，可联系芯祥联科技技术支持：hezuo@xxltech.cn，或访问公司官网www.xxltech.cn获取更多合作信息。

（注：文档部分内容可能由 AI 生成）