在 MA35D1 上实现实时视频流

NuMicro® 64 位系列微控制器范例代码介绍

文件信息

应用简述	在 MA35D1 FreeRTOS 平台上，使用 Real-time Transport Protocol 传输视频流，并利用 MA35D1 上的 H.264 decoder 解码并播放。
BSP 版本	MA35D1_NonOS_BSP V1.4
开发平台	NuMaker-HMI-MA35D1-S1 V1.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. 概述

本示例程序展示如何在 Nuvoton MA35D1 FreeRTOS 平台上，结合 lwIP 网络协议栈与 RTP (Real-time Transport Protocol) 技术，实现高性能的视频流实时解码与显示系统。与传统使用 RTSP 协议不同，本方案采用更为轻量直接的 UDP Server 接收方式，跳过信令协商流程，直接处理来自 IPCAM 的 RTP 数据包，这对于追求极低延迟的嵌入式视频监控应用至关重要。

在实现机制上，系统启动 FreeRTOS 后会立即初始化 MA35D1 内置的 H.264 硬件解码器。为了确保网络数据包在高码率传输下的稳定性，本示例特别引入了 Ring Buffer (环形缓冲区) 处理机制：从网络层接收到的 RTP 数据会先暂存于 Buffer 中，经由程序解包、重组后，直接送入硬件解码引擎进行处理并输出至显示接口。这种设计充分发挥了 MA35D1 的硬件加速能力，即便在 RTOS 环境下，也能提供流畅且稳定的视频播放体验。

此示例程序是基于 MA35D1_NonOS_BSP V1.4 开发环境（https://github.com/OpenNuvoton/MA35D1_NonOS_BSP）进行开发的。在开始执行前，请确保您已搭建好对应的开发环境，并获取相关驱动程序库。

虽然本文是以 NuMaker-HMI-MA35D1-S1 V1.0 开发板作为实现演示，但其内容与核心代码适用于所有具备 H.264 硬件解码功能的 MA35D1 系列 MPU。对于开发工业监控终端、智能安防显示器或任何需要高效视频流处理的嵌入式产品，均具有极高的参考价值。

1.1 原理

本系统旨在 FreeRTOS 实时操作系统环境下，针对嵌入式视频监控与智能显示终端，提供极低延迟的 H.264 视频流解决方案。系统核心整合了 lwIP 网络协议栈与 MA35D1 内置的硬件解码引擎，其主要运行原理可分为以下几个技术维度：

1.1.1 轻量化低延迟传输：基于 UDP 的 RTP 直连解析

与传统依赖 RTSP (Real-Time Streaming Protocol) 的方案不同，本系统在架构上进行了大幅度的轻量化。

- 免除协议握手开销： 传统 RTSP 需经历复杂的连接握手与状态控制（如 DESCRIBE、SETUP、PLAY 等），容易造成视频显示的首帧延迟 (First Frame Out Delay)。本方案直接部署 UDP Server 来接收 RTP (Real-time Transport Protocol) 数据包，跳过所有前端信令协商。

- 实时数据包处理： 来自 IPCAM 的视频数据通过 RTP 数据包形式抵达后，系统会直接通过 lwIP 提取 UDP 负载（Payload），并解析 RTP 报头以获取序列号与时间戳，确保实时视频的时序正确性。此策略极大地缩短了网络端到应用端的数据传递路径。

1.1.2 稳定数据流控：Ring Buffer 环形缓冲机制

在工业应用与高清监控场景中，高码率（High Bitrate）的 H.264 视频流容易因瞬间网络波动或 RTOS 任务切换产生突发数据量。为确保解码稳定性，系统引入了 Ring Buffer 机制。

- 异步生产与消费： 网络接收任务（Producer）将解包后的 RTP 视频负载快速写入 Ring Buffer；而解码任务（Consumer）则依据硬件引擎的处理节奏，从 Buffer 中读取完整的 H.264 视频帧（Frame）数据。
- 防抖动与平滑化： 通过此环形缓冲区，系统能有效吸收网络传输过程中的 Jitter（延迟抖动），避免硬件解码器因数据匮乏（Underflow）而产生画面卡顿，或因数据溢出（Overflow）导致丢帧。

1.1.3 高性能视频处理：MA35D1 核心硬件解码加速

在解开 RTP 数据包并重组出标准的 H.264 网络抽象层单元 (NALU, Network Abstraction Layer Unit) 后，数据会直接交由 MA35D1 的硬件解码器进行处理。

- 卸载 CPU 运算压力： 将繁重的视频解码工作交由专属硬件引擎执行，释放了 Cortex-A35 核心的运算资源，使其能专注于 FreeRTOS 的实时任务调度、网络数据包收发及其他 HMI 控制逻辑。
- 直通显示接口： 硬件解码器输出 YUV 或 RGB 原始视频数据后，可无缝衔接 MA35D1 的显示控制器（Display Controller），直接输出至 LCD 面板，完成从网络端到视觉终端的高效数据流转。

1.1.4 整体数据流流程 (Data Flow Pipeline)

综上所述，系统的标准处理流水线（Pipeline）如下：

- 接收： lwIP 于 FreeRTOS 中监听 UDP 端口，接收来自 IPCAM 的 RTP 数据包。
- 解包与缓冲： 系统去除 UDP/RTP 报头，提取 H.264 NALU 数据并依序推入 Ring Buffer。

- 硬件解码： 解码任务自 Ring Buffer 提取完整视频帧，驱动 MA35D1 H.264 硬件解码引擎进行实时解码。
- 渲染输出： 解码后的高帧率视频直接推送至显示接口完成画面更新。

1.2 环境设置

本示例程序以 C 语言编写，并运行于 FreeRTOS 实时操作系统环境中，执行过程中会高度依赖底层硬件寄存器设置与 lwIP 网络协议栈。因此，在开始编译与执行前，必须先确认相关的集成开发环境 (IDE)、板级支持包 (BSP) 以及烧录工具均已正确安装并设置完毕。

由于 Nuvoton 官方提供的工具链已针对 MA35D1 平台进行了完善的集成，本说明将以原厂标准工具作为开发与执行环境。您需要准备并确认以下开发工具：

- 集成开发环境 (IDE) - NuEclipse （V1.02.029）：

  本示例的项目构建、程序编译与调试均使用 NuEclipse 进行。关于 IDE 详细安装步骤与操作指南，请参考 NuEclipse User Manual。

- 板级支持包 (BSP) - MA35D1 Non-OS BSP：
- 系统底层的 H.264 驱动与网络控制器操作皆基于官方 Non-OS BSP (包含 FreeRTOS 内核)。请确保您已自 GitHub 项目页面获取对应版本。关于 BSP 的架构说明与 API 使用方式，可参考 NuMicro® Family MA35D1 Non-OS BSP User Manual。
- 镜像文件烧录工具 - NuWriter （V1.08）：

  当代码编译完成并产出可执行文件 (Image) 后，需通过 NuWriter 工具，将镜像文件烧录至开发板的存储介质中（如 SD Card、NAND Flash 或 eMMC）或 DDR 执行模式以供开机执行。

1.2.1 导入 NuEclipse 项目

打开 NuEclipse，再依下列步骤导入本示例：

File -> Import ->General -> Existing Projects into Workspace -> Select root directory -> Finish

导入完成后，右键点击该项目并选择 Build Project 即可开始编译。

1.3 烧录与执行结果

在开始编译之前，请前往以下路径：

MA35D1_NonOS_BSP/Library/VC8000Lib

请将目录下的 libvc8000_RTOS.a 重新命名，并覆盖原有的 libvc8000.a 文件。此文件是专为 FreeRTOS 环境进行优化的驱动程序函数库。

译完成后，产生的可执行文件 (.bin) 将会输出于以下路径：

MA35D1_NonOS_BSP\SampleCode\lwIP\MA35D1_IPCam_RTP\GCC\Release\MA35D1_IPCam_RTP.bin

本示例将使用 NuWriter 工具将程序下载至 DDR 中直接执行。请依照以下步骤进行操作：

1. 连接硬件接口：请确保开发板已正确连接必要的线路，包括调试端口 (Debug port，用于终端信息输出)、USB 设备端 (USBD，供 NuWriter 连接下载使用)，以及以太网 (Ethernet，用于接收 RTP 视频流)。

2. 开启开发模式：启动 NuWriter 后，点击上方菜单的 Mode -> Develop mode 以开启开发模式。

3. 连接开发板：切换至 Attach 页面，在下拉菜单中选择对应开发板型号的 Part Number，接着点击 Attach 按钮建立连接。

4. 设置下载参数：连接成功后（参考图 1-1），切换至 Download 页面，并选择 DDR/SRAM 选项卡。

5. 加载并执行：点击浏览并选择上述编译完成的可执行文件位置，确保执行地址 (Execute Address) 设置为 0x80400000，最后点击 Write 完成程序下载与执行。

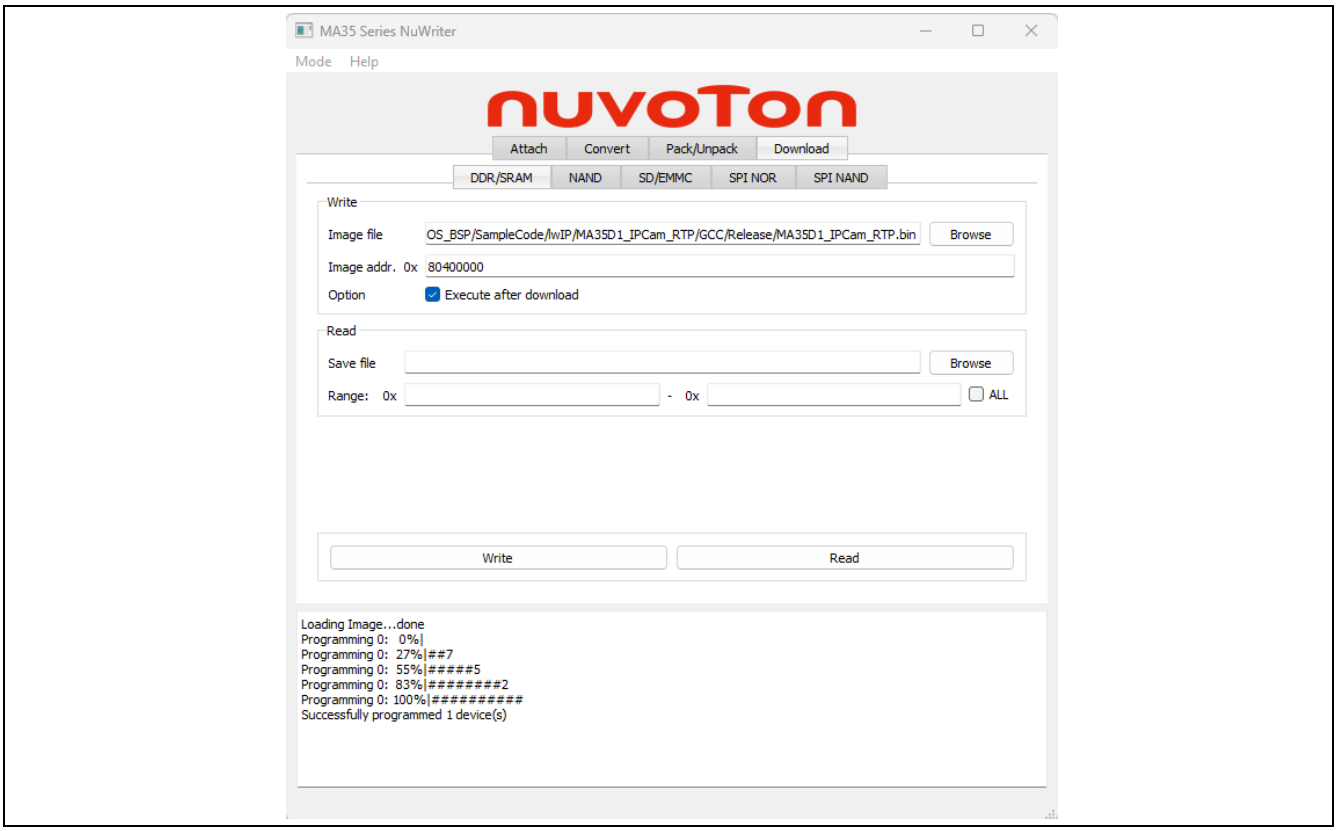


图 1-1NuWriter 烧录画面

烧录完成后，可以从终端看到如图 1-2 所示的调试信息，后续即可开始接收网络数据包进行解码并播放视频。

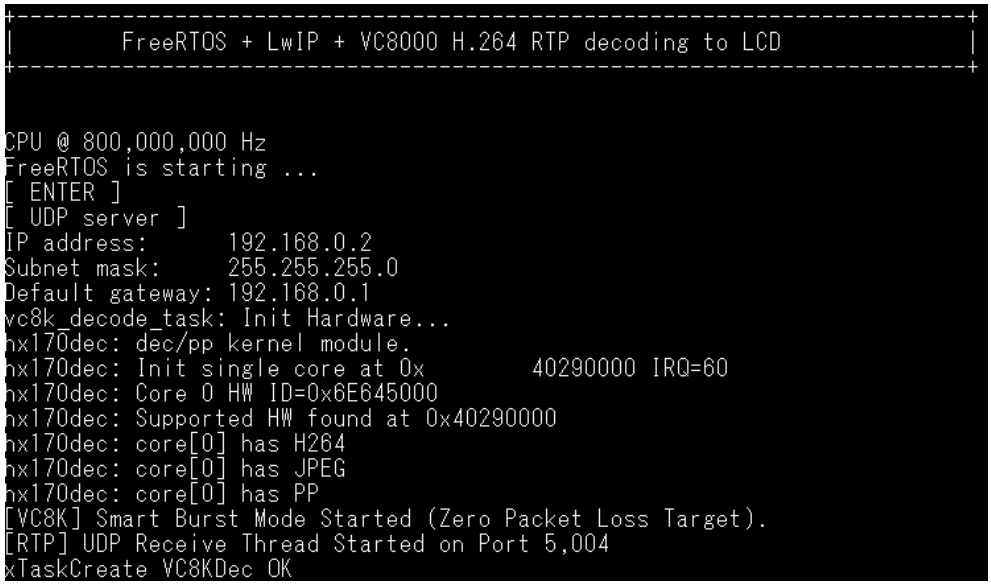


图 1-2 终端画面

2. 代码介绍

2.1 main.c

这个 main.c 程序是整个项目的系统总指挥 (Entry Point)。它的主要任务是在 Nuvoton MA35D1 微处理器上，建立一个基于 FreeRTOS 的「实时视频流接收与播放系统」。

具体来说，它负责严密地协调以下三大硬件与软件模块的运行：

- 视觉输出 (Display)： 在系统开机时，初始化系统时钟与 LCD 显示控制器，并规划好 Frame Buffer (帧缓冲区)，确保屏幕能正常显示开机 Logo 与后续的动态视频。
- 网络接收 (Network)： 启动 lwIP 网络协议栈，设置开发板的 IP 地址，并建立一个 UDP 服务器。这个服务器专门用来接收来自远程 (如 IP Camera) 的 RTP (H.264) 视频数据包，并将其推入 Ring Buffer 缓冲区。
- 硬件解码 (Decoding)： 配置好专属的内存空间后，唤醒 VC8000 硬件解码任务。该任务会不断从缓冲区提取视频数据，交由硬件加速解码，最后直接输出至屏幕。

2.1.1 系统硬件内存布局 (Memory Layout)

在处理高清视频解码时，精确的内存配置至关重要。由于 MA35D16A887C 具备 256 MB 的 DDR 内存，本程序代码开头即明确定义了各个硬件加速单元专属的物理内存地址。

系统将 0x80000000 前段留给代码与 FreeRTOS 堆栈使用，并将 0x88000000 之后的连续大空间严格划分给 LCD Frame Buffer (显示缓冲)、RTP Ring Buffer (1 MB 网络接收缓冲) 以及 VC8000 硬件解码器专属的 Heap (32 MB)。这种直接指定物理地址的静态划分方式，能有效避免 RTOS 运行期间因频繁分配大内存而产生的碎片化或溢出风险。

```
/* ---------------- Memory Layout ----------------- */
/*
 * DDR Range: 0x80000000 ~ 0x90000000 (256 MB)
 * 0x80000000 : Code / BSS / FreeRTOS Heap
 * 0x88000000 : Frame Buffer (Start)
 * 0x89000000 : RTP Ring Buffer (Start, 1 MB)
 * 0x8B000000 : VC8000 Heap (Start, 32 MB)
 */
#define DDR_ADR_FRAMEBUFFER    0x88000000UL
#define RTP_RING_ADDR          0x89000000UL
#define RTP_RING_BUFFER_SIZE   (1 * 1024 * 1024)
```

2.1.2 网络与解码任务桥接配置 (vUdpTask)

vUdpTask 是串接「网络接收」与「硬件解码」两大系统核心的关键任务。程序首先通过 lwIP 协议栈设置开发板的静态 IP 地址 (192.168.0.2) 并启用网络接口。接着，将 RTP Ring Buffer 初始化并绑定至预先规划的 DDR 地址上，同时启动负责接收视频数据包的 UDP Server。最后，通过 xTaskCreate 创建 vc8k_decode_task 任务，确立了由 UDP 网络端作为「生产者」，VC8000 硬件解码端作为「消费者」的高效异步架构。

```
/* vUdpTask: set IP, create RTP receiver, and create VC8K decode task */
static void vUdpTask( void *pvParameters )
{
    ip_addr_t ipaddr;
    ip_addr_t netmask;
    ip_addr_t gw;

    (void) pvParameters;

    IP4_ADDR(&ipaddr,   192,168,0,2);
    IP4_ADDR(&netmask,  255,255,255,0);
    IP4_ADDR(&gw,       192,168,0,1);

    tcpip_init(NULL, NULL);

    netif_add(&netif, &ipaddr, &netmask, &gw, NULL,
              ethernetif_init(GMAC_INTF), tcpip_input);
    netif_set_default(&netif);
    netif_set_up(&netif);

    sysprintf("[ UDP server ] \n");
    sysprintf("IP address:      %s\n", ip4addr_ntoa(&netif.ip_addr));
    sysprintf("Subnet mask:     %s\n", ip4addr_ntoa(&netif.netmask));
    sysprintf("Default gateway: %s\n", ip4addr_ntoa(&netif.gw));

    /* Initialize Ring Buffer with fixed address */
    /* Direct cast without nc_ptr macro */
    rtp_ring_init(&g_rtp_ring,
                  (uint8_t *)g_rtp_ring_buf,
                  RTP_RING_BUFFER_SIZE);

    /* Start RTP receive thread */
    udp_echoserver_netconn_init();

    /* Create VC8K Decoder Task */
    BaseType_t rc = xTaskCreate(vc8k_decode_task,
                                "VC8KDec",
                                4096,            /* 16 KB stack */
                                NULL,
                                VC8K_DECODE_PRIO,
                                NULL);

    if (rc != pdPASS)
        sysprintf("xTaskCreate VC8KDec FAILED! rc=%d\n", (int)rc);
```

```
    else
        sysprintf("xTaskCreate VC8KDec OK\n");

    vTaskSuspend(NULL);
}
```

2.1.3 主程序系统与显示接口初始化 (main)

main 函数为系统启动的总控制中心。在完成底层时钟与硬件初始化 (SYS_Init) 后，程序会先进行开机画面的处理：通过指针运算计算开机 Logo 的大小，并利用 memcpy 将图像数据直接拷贝至 DDR 中的 Frame Buffer (帧缓冲区) 地址。随后调用 DISPLIB_SetFBConfig，将 LCD 显示控制器精准导向该 Frame Buffer 地址并启用视频层输出。一切硬件就绪后，便创建 UdpTask 任务并正式启动 FreeRTOS 系统调度器，让系统从单线程进入强大的多任务运行阶段。

```
int main(void)
{
    SYS_Init();
    sysprintf("\n+------------------------------------------------------------------------
+\n");
    sysprintf("|        FreeRTOS + LwIP + VC8000 H.264 RTP decoding to LCD
|\n");
    sysprintf("+------------------------------------------------------------------------
+\n");

    /* Display boot logo */
    uint32_t file_size;
    DISP_Open();
    DISPLIB_DDR_AXIPort_Priority();

    /* Calculate logo size using safe pointer conversion */
    file_size = PTR2U32(&ImageDataLimit) - PTR2U32(&ImageDataBase);

    /* Copy boot logo to Frame Buffer */
    memcpy((void *)DDR_ADR_FRAMEBUFFER,
           (void *)&ImageDataBase,
           file_size);

    /* Initialize Display with Frame Buffer */
    DISPLIB_LCDInit(LcdPanelInfo);
    DISPLIB_SetFBConfig(eFBFmt_A8R8G8B8,
                        LcdPanelInfo.u32ResolutionWidth,
                        LcdPanelInfo.u32ResolutionHeight,
                        DDR_ADR_FRAMEBUFFER);
    DISPLIB_EnableOutput(eLayer_Video);

    /* FreeRTOS + LwIP + VC8K */
    global_timer_init();
    sysprintf("\n\nCPU @ %d Hz\n", SystemCoreClock);
    sysprintf("FreeRTOS is starting ...\n");
    xTaskCreate(vUdpTask, "UdpTask",
                UDP_THREAD_STACKSIZE,
                NULL,
                UDP_TASK_PRIORITY,
```

```
                NULL);
    vTaskStartScheduler();
    for (;;);
}
```

2.2 rtp_h264.c

这份 rtp_h264.c 的核心功能是将网络层接收到的 RTP 数据包，转换为硬件解码器可识别的 H.264 Annex-B 码流格式（即带有 00 00 00 01 起始码的 NALU）。

2.2.1 主解析入口与 NAL 分流机制 (h264_rtp_to_annexb)

h264_rtp_to_annexb 为此模块的主要入口函数。它首先会严格验证 RTP 头部 (Header) 的合法性，并处理 Extension 与 Padding 字段，确保精准定位到 H.264 Payload (负载)。特别重要的是，此处加入了防御性设计：主动丢弃对硬件解码无用的 SEI (Type 6) 与 AUD (Type 9) 数据包，借此保护后端解析器并节省内存带宽。最后再根据 NAL 类型，将数据分流给对应的处理函数。

```
/* Determine NAL Type */
uint8_t nal0 = h264_payload[0];
uint8_t nal_type = H264_NAL_TYPE(nal0);

/* Defense: Drop useless SEI (6) and AUD (9) to protect parser and save memory bandwidth
*/
if (nal_type == 6 || nal_type == 9) {
    return 0;
}

int ret = 0;

switch (nal_type)
    {
    case 1 ... 5:
    case 7:
    case 8:
    case 10 ... 23:
        /* Single NAL Unit Packet */
        ret = handle_single_nal(ring, h264_payload, h264_len);
        break;

    case 24:
        /* STAP-A (Aggregation Packet) */
        ret = handle_stap_a(ring, h264_payload, h264_len);
        break;

    case 28:
        /* FU-A (Fragmentation Unit A) */
        ret = handle_fu_a(ring, h264_payload, h264_len);
        break;

    default:
        /* Unsupported types (STAP-B, MTAP, FU-B) are safely ignored */
```

```
            ret = 0;
            break;
}
```

2.2.2 FU-A 分片重组与容错机制 (handle_fu_a)

当 H.264 视频帧（如 I-Frame）过大超过网络 MTU 时，会被切分成多个 RTP 数据包，称为 FU-A。此函数通过解析 S (Start) 与 E (End) 位来还原数据。遇到 Start 时，会重建原始的 NAL Header 并加上 Annex-B 起始码。为应对 UDP 丢包风险，程序使用了 s_in_fu_a 全局变量进行状态跟踪；若丢失了 Start 数据包，后续的分片（孤儿分片）将被安全地丢弃，避免将残缺的帧数据送入解码器从而导致系统崩溃或画面花屏。

```
uint8_t S = (uint8_t)(fu_header & 0x80U); /* Start bit */
uint8_t E = (uint8_t)(fu_header & 0x40U); /* End bit */

/* Reconstruct NAL header */
uint8_t reconstructed_nal = (uint8_t)(F | NRI | nal_type);

if (S)
{
    /* Mark that a new frame fragment has correctly started */
    s_in_fu_a = true;

    /* Start fragment: Output start code + Reconstructed NAL Header + Data */
    // ... (Push start code, reconstructed NAL, and frag_data to ring buffer)
}
else
{
    /* Defense: Drop orphaned fragments if the start bit was previously missed */
    if (!s_in_fu_a) return 0;

        /* Middle or End fragment: Append data only */
        // ... (Push frag_data to ring buffer)
    }

    /* Reset tracking state upon receiving the end fragment */
    if (E) {
        s_in_fu_a = false;
    }
```

2.3 rtp_receiver.c

本代码 (rtp_receiver.c) 在整个实时视频流系统中扮演「数据生产者」的角色，专责处理底层网络收发。它利用 lwIP 的 Netconn API 建立了一个独立的 FreeRTOS 网络接收任务，持续监听指定的 UDP 端口 (默认为 5004)。当接收到来自远程 (如 IP Camera) 的 RTP 视频数据包时，系统会将数据包内容提取出来，并交由后端的 H.264 解析模块进行处理，是串接物理网络与视频解码引擎的第一道关键桥梁。

2.3.1 网络接收任务 (rtp_udp_recv_thread)

此函数是负责实际接收网络数据包的核心线程。为了满足高清视频频繁收发的严苛要求并确保系统长时间运行的稳定性，此函数具备以下几个关键设计：

- 静态内存分配： 特别声明了固定大小的静态数组 (payload_buf) 作为接收缓冲区，彻底避开了因频繁调用 malloc/free 所引发的性能损耗与内存碎片化 (Memory Fragmentation) 风险。
- 阻塞式等待与安全防护： 线程以阻塞模式 (netconn_recv) 挂起等待数据包，避免无谓的 CPU 资源消耗。收到数据包后，会优先进行长度检查 (len > RTP_MAX_UDP_PAYLOAD)，主动丢弃过大的异常数据包以防范缓冲区溢出。
- 防止资源泄漏： 在完成数据包的复制与解析调用后，程序必定会执行 netbuf_delete(buf) 来释放 lwIP 内部的数据链路块，这是防止系统发生内存泄漏 (Memory Leak) 的关键步骤。

```
static void rtp_udp_recv_thread(void *arg)
{
    /* Prevent compiler warning for unused argument */
    (void)arg;

    struct netconn *conn;
    struct netbuf  *buf;
    err_t           err;

    /* Fixed receive buffer: avoids malloc/free overhead on every packet */
    static uint8_t payload_buf[RTP_MAX_UDP_PAYLOAD];

    sysprintf("[RTP] UDP Receive Thread Started on Port %d\n", RTP_UDP_PORT);

    /* Create a new UDP connection handle */
    conn = netconn_new(NETCONN_UDP);
    if (conn == NULL)
    {
        sysprintf("[RTP] Error: netconn_new failed\n");
        vTaskDelete(NULL);
    }

    /* Bind to the specific RTP port */
    err = netconn_bind(conn, NULL, RTP_UDP_PORT);
    if (err != ERR_OK)
    {
        sysprintf("[RTP] Error: netconn_bind failed (err=%d)\n", err);
        netconn_delete(conn);
        vTaskDelete(NULL);
    }

    while (1)
```

```
    {
        /* Blocking receive: Task suspends here until a packet arrives */
        err = netconn_recv(conn, &buf);

        if (err != ERR_OK)
        {
            /* Check if connection was closed or other fatal errors */
            sysprintf("[RTP] netconn_recv error: %d\n", err);
            continue;
        }

        /* Get payload length */
        u16_t len = netbuf_len(buf);

        /* Safety check: Drop oversized packets to prevent buffer overflow */
        if (len > RTP_MAX_UDP_PAYLOAD)
        {
            /* In release mode, silent drop or minimal logging is preferred */
            // sysprintf("[RTP] Drop oversized packet: %u bytes\n", len);
            netbuf_delete(buf);
            continue;
        }

        /* Copy data from LwIP chain to linear buffer */
        netbuf_copy(buf, payload_buf, len);

        /* Process H.264 RTP payload -> Annex-B -> Ring Buffer */
        int ret = h264_rtp_to_annexb(&g_rtp_ring, payload_buf, len);

        if (ret < 0)
        {
            /* Error handling:
             * -1: Malformed packet
             * -2: Ring buffer full (Overflow)
             * -3: Push error
             */
            // sysprintf("[RTP] Parse error or Ring full: %d\n", ret);
        }

        /* Important: Free the LwIP buffer to prevent memory leaks */
        netbuf_delete(buf);
    }
}
```

2.4 rtp_ring.c

本代码实现了针对硬件视频解码优化的进阶环形缓冲区 (Ring Buffer)。传统环形缓冲区在写入指针到达边界时会产生回绕 (Wrap-around) 现象，导致单一 H.264 视频帧数据在内存地址上不连续。由于 MA35D1 内置的 VC8000 硬件解码引擎依赖 DMA 进行数据访问，这要求物理内存地址必须绝对连续。

为解决地址不连续的限制并最大化系统性能，本模块引入了「隐藏备用区 (Hidden Headroom Backup)」机制。通过在缓冲区前端保留特定大小的内存块，当发生数据回绕时，系统能快速将碎片化数据拼接为连续区块，实现零拷贝线性化 (Zero-Copy Linearization)，有效免除额外分配临时缓冲区与整包数据拷贝的庞大系统开销。

2.4.1 零拷贝线性化 API (rtp_ring_peek_linear)

此函数为本模块的核心，其功能是向硬件解码任务提供「物理地址绝对连续」的视频数据指针。其底层运行机制分为以下两种场景：

- 场景一（物理地址连续）：

  若当前写入指针 (wr) 位于读取指针 (rd) 之后，表示数据未跨越缓冲区边界，具备物理连续性。此时系统不进行任何内存搬移，直接返回当前地址与可读长度，无额外 CPU 运算开销。

- 场景二（回绕线性拼接）：

  若数据发生回绕，导致完整视频帧被分割为「尾段 (Tail)」与「头段 (Head)」。系统会将尾段数据复制至预先保留的 128 KB 隐藏备用区，使其与头段数据在物理地址上完美衔接。同时，系统重定向读取指针至备用区，将原本碎片化的数据转化为单一连续指针供解码器读取。

```
/**
 * @brief   【Core Linearization API】
 * Obtain a guaranteed physically contiguous memory pointer.
 * If a wrap-around occurs, it automatically uses the backup area to align the data.
 */
uint32_t rtp_ring_peek_linear(RTP_RING *r, uint8_t **ptr)
{
    uint32_t rd = r->rd;
    uint32_t wr = r->wr;
    uint32_t ring_end = PREALLOC_BACKUP_SIZE + r->size;

    /* Condition 1: No wrap-around, physical memory is inherently continuous */
    if (wr >= rd)
    {
        *ptr = r->buf + rd;
        return wr - rd;
    }

    /* Condition 2: Wrap-around occurred. The tail has a segment, and the head has a
segment */
    uint32_t tail_len = ring_end - rd;

    /* If the tail is smaller than the backup area, copy it backwards */
    if (tail_len <= PREALLOC_BACKUP_SIZE)
    {
```

```
        uint32_t backup_start = PREALLOC_BACKUP_SIZE - tail_len;

        /* Move the tail data to the backup area to align with (buf +
PREALLOC_BACKUP_SIZE) */
        memcpy(r->buf + backup_start, r->buf + rd, tail_len);

        /* Relocate the read pointer into the backup area */
        r->rd = backup_start;
        *ptr = r->buf + r->rd;

        /* Return the combined total length: tail length + head length */
        return tail_len + (wr - PREALLOC_BACKUP_SIZE);
    }

    /* Extreme failsafe: The tail is too large for the backup area, return only the tail
*/
    *ptr = r->buf + rd;
    return tail_len;
}
```

2.5 vc8k_decode.c

本代码 (vc8k_decode.c) 实现了驱动 MA35D1 内置 VC8000 硬件解码引擎的 FreeRTOS 独立任务，在系统中扮演「数据消费者」的角色。其主要职责是从网络接收任务写入的环形缓冲区 (Ring Buffer) 中提取 H.264 视频流，并将其送入硬件引擎进行高效解码。解码完成后，通过硬件内置的后处理器 (Post-Processor, PP)，系统会将图像直接转换为 LCD 面板所需的色彩格式并输出至显示缓冲区 (Frame Buffer)。

2.5.1 硬件解码任务与后处理初始化 (vc8k_decode_task)

此函数为解码线程的入口点，负责完成 VC8000 核心的硬件初始化与专属内存 (Heap) 分配。其中最具技术价值的设计在于「硬件后处理器 (Post-Processor)」的配置：程序会依据 LCD 面板的分辨率，直接将硬件解码后的输出格式设置为 VC8000_PP_F_RGB888，并将目标物理地址 (pp_out_paddr) 绑定至系统保留的 Frame Buffer 地址 (VC8K_FB_ADDR)。这项设置确保了图像一旦解码完毕，硬件便会自动完成 YUV 到 RGB 的色彩空间转换并写入显示内存。

```
void vc8k_decode_task(void *arg)
{
    (void)arg;
    int ret, handle;
    struct pp_params pp;

    sysprintf("vc8k_decode_task: Init Hardware...\n");

    /* Initialize VC8000 with Heap Memory */
    ret = VC8000_Init(VC8K_HEAP_ADDR, VC8K_HEAP_SIZE);
    if (ret < 0) vTaskDelete(NULL);

    VC8000_InstallH264DecCallback(vc8k_h264_done_cb);
```

```
    handle = VC8000_H264_Open_Instance();
    if (handle < 0) vTaskDelete(NULL);

    /* Configure Post-Processor (PP) for Display */
    memset(&pp, 0, sizeof(pp));
    pp.frame_buf_w = LcdPanelInfo.u32ResolutionWidth;
    pp.frame_buf_h = LcdPanelInfo.u32ResolutionHeight;
    pp.img_out_w   = pp.frame_buf_w;
    pp.img_out_h   = pp.frame_buf_h;
    pp.img_out_fmt = VC8000_PP_F_RGB888;
    pp.pp_out_dst  = VC8000_PP_OUT_DST_USER;
    pp.pp_out_paddr = VC8K_FB_ADDR;

    ret = VC8000_H264_Enable_PP(handle, &pp);
    if (ret < 0) {
        VC8000_H264_Close_Instance(handle);
        vTaskDelete(NULL);
    }

    /* Enter Decoding Loop */
    VC8000_BitstreamDecodeLoop(handle, &g_rtp_ring);

    VC8000_H264_Close_Instance(handle);
    vTaskDelete(NULL);
}
```

2.5.2 智能解码循环与缓存同步机制 (VC8000_BitstreamDecodeLoop)

此函数是持续将数据送入硬件解码引擎的核心无限循环。为了确保 RTOS 性能与硬件稳定性，本函数引入了几个关键机制：

1. 智能突发模式 (Smart Burst)： 当缓冲区积累的数据量超过阈值 (MIN_BURST_SIZE = 4 KB) 时才唤醒硬件处理，有效避免因处理微小数据包而产生频繁的 RTOS 上下文切换 (Context Switch) 开销。

2. 缓存一致性管理 (Cache Flush)： 由于 VC8000 是通过 DMA 直接读取外部内存，程序在调用硬件解码前，必定会执行 dcache_clean_by_mva。此操作能将 CPU 暂存在 Cache (缓存) 中的图像数据强制刷新至物理 DDR 中，防止硬件读取到未更新的旧数据从而导致画面花屏。

3. 动态缓冲区更新： 硬件执行完毕后，会返回实际未消耗的字节数 (remain)，程序据此精准计算出硬件实际消耗的数据量 (consumed)，并同步更新 Ring Buffer 的读取指针，维持系统数据流的稳定平衡。

```
int VC8000_BitstreamDecodeLoop(int handle, RTP_RING *ring)
{
    uint8_t  *bs_ptr;
    uint32_t  bs_len;
```

```
    uint32_t  remain;
    int       ret;

    /* Smart Burst Configuration
     * Threshold set to 4 KB to ensure sufficient data for hardware
     * while avoiding excessive context switching for small packets.
     */
    const uint32_t MIN_BURST_SIZE = 4 * 1024;
    sysprintf("[VC8K] Smart Burst Mode Started (Zero Packet Loss Target).\n");

    /* Statistics variables */
    static uint32_t last_frame_count = 0;
    static uint32_t last_tick = 0;

    /* Initialize tick count */
    last_tick = xTaskGetTickCount();
    for (;;)
    {
        /* --- 0. Status Monitor (Print every 300 frames) --- */
        if (s_h264_frame_count >= last_frame_count + 300)
        {
            uint32_t now = xTaskGetTickCount();
            uint32_t diff_time = now - last_tick;
            uint32_t diff_frames = s_h264_frame_count - last_frame_count;

            /* Convert ticks to ms (Assuming 1 Tick = 1ms) */
            uint32_t ms_total = diff_time;

            if (ms_total > 0)
            {
                uint32_t fps = (diff_frames * 1000) / ms_total;
                uint32_t ms_per_frame = ms_total / diff_frames;

                sysprintf("[Perf] Speed: %lu FPS | Cost: %lu ms/frame | RingUsed: %u\n",
                          fps, ms_per_frame, rtp_ring_used(ring));
            }

            last_frame_count = s_h264_frame_count;
            last_tick = now;
        }

        /* --- 1. Check available data --- */
        uint32_t used = rtp_ring_used(ring);

        if (used < MIN_BURST_SIZE)
        {
            vTaskDelay(pdMS_TO_TICKS(1));
            continue;
        }

        /* --- 2. Obtain physically contiguous memory ---
         * rtp_ring_peek_linear ensures the returned bs_ptr is a contiguous
         * block, automatically handling boundary wrap-arounds via the backup area.
         */
        bs_len = rtp_ring_peek_linear(ring, &bs_ptr);

        /* Limit the feed size to the actual used data to prevent over-reading */
        if (bs_len > used)
```

```
        {
            bs_len = used;
        }

        /* --- 3. Cache Flush (Critical) --- */
        dcache_clean_by_mva((void *)bs_ptr, bs_len);

        remain = bs_len;

        /* --- 4. Hardware Decode Run --- */
        ret = VC8000_H264_Decode_Run(handle, (uint8_t *)bs_ptr, bs_len, NULL, &remain);

        /* Filter out false alarm (-201): PP init rejected due to identical SPS repetition
*/
        if (ret == -201)
        {
            ret = 0;
        }

        if (ret < 0)
        {
            sysprintf("[VC8K ERR] ret = %d, bs_len = %lu, used = %lu\n", ret, bs_len,
used);
            /* Skip the entire broken chunk to search for the next NALU */
            rtp_ring_consume(ring, bs_len);
            continue;
        }

        /* --- 5. Update Ring Buffer --- */
        if (remain > bs_len) remain = bs_len;
        uint32_t consumed = bs_len - remain;
        if (consumed == 0)
        {
            /* Hardware didn't consume data (needs more headers/data) */
            vTaskDelay(pdMS_TO_TICKS(1));
            continue;
        }

        rtp_ring_consume(ring, consumed);

        /* --- 6. Greedy Scheduling Strategy --- */
        /* If there is still a lot of data in the buffer (e.g., due to wrap-around
         * or fast network burst), continue immediately without yielding.
         */
        if (rtp_ring_used(ring) > MIN_BURST_SIZE)
        {
            continue;
        }
        else
        {
            /* Buffer is low, yield to allow other tasks to run */
            taskYIELD();
        }
    }
}
```

3. 软件与硬件需求

3.1 软件需求

- BSP 版本
  - MA35D1_NonOS_BSP V1.4
  - NuWriter V1.08
  - NuEclipse V1.02.029
- Third Party Simulation Tool
  - FFmpeg version 4.2.7-0ubuntu0.1

3.2 硬件需求

- 电路元件
  - NuMaker-HMI-MA35D1-S1 V1.0

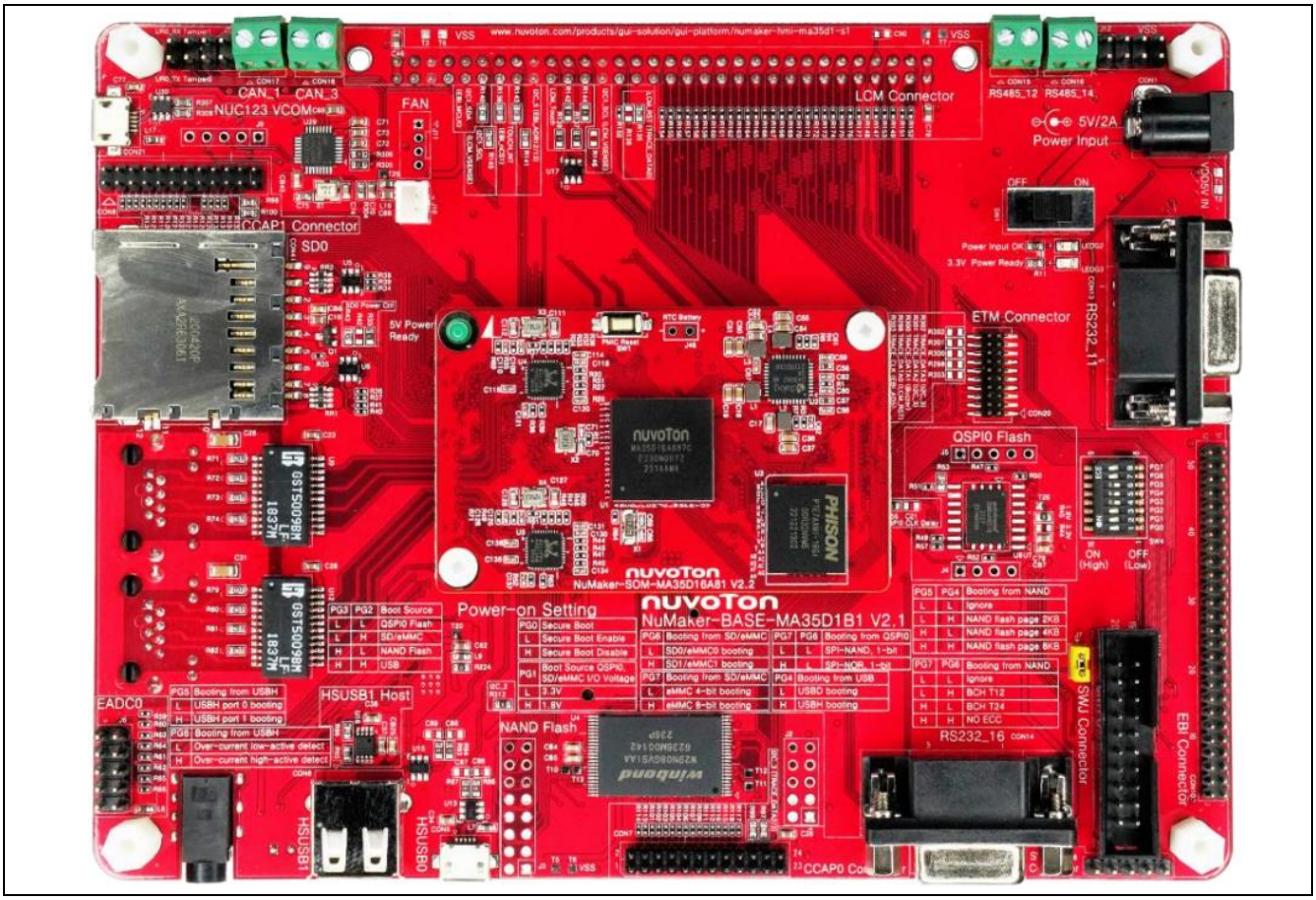


图 3-1 NuMaker-HMI-MA35D1-S1 V1.0

4. 目录信息

目录结构如下图所示

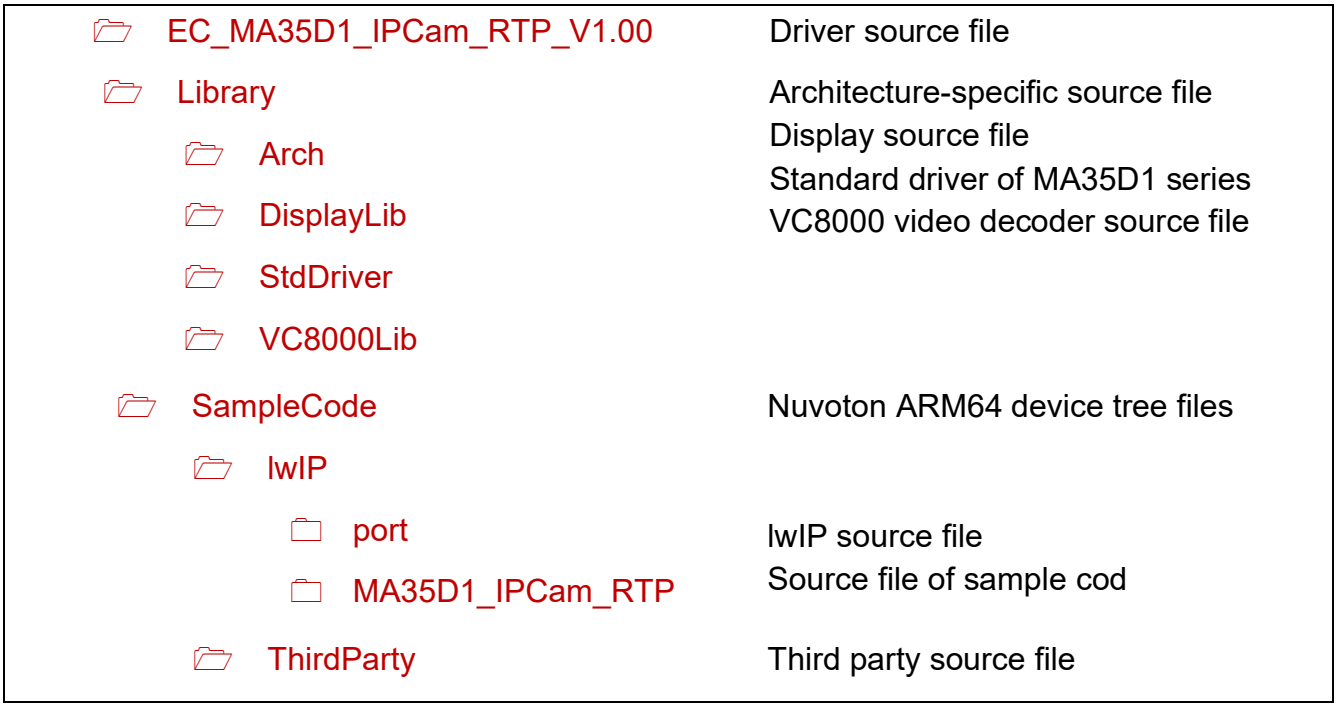


图 4-1 目录信息

5. 示例程序运行

本示例程序在如**錯誤! 找不到參照來源。**章节烧录完成后，会开始监听并等待接收 H.264 RTP 视频流数据包。要将视频数据包发送至开发板的 IP 地址，您可以使用实体的 IP Camera，或是在 PC 端通过命令模拟发送。

以下说明如何利用 PC（以 Ubuntu 系统为例）上的 FFmpeg （version 4.2.7-0ubuntu0.1）工具来模拟 IPCAM 发送 RTP 数据包：

1. 准备测试视频： 请准备一段 H.264 格式的测试视频（本示例使用 1024600.mp4），并将该视频文件放置于 Ubuntu 系统准备执行命令的当前文件夹路径中。

2. 执行 FFmpeg 推流命令： 打开 Ubuntu 终端 (Terminal)，输入以下命令以原生帧率读取视频，并通过 RTP 协议发送至开发板。

```
user@ubuntu:~/temp$ ffmpeg -re -stream_loop -1 -i 1024600.mp4 -c:v libx264 -preset
veryfast -tune zerolatency -pix_fmt yuv420p -an -x264-params slice-max-size=1300 -f rtp -
rtpflags h264_mode0 rtp://192.168.0.2:5004
```

3. 确认推流状态： 命令执行后，Ubuntu 终端将会持续打印出当前的推流进度与传输速率，PC 端的画面请参考图 5-1。此时，开发板端应已成功接收 RTP 数据包，并开始进行硬件解码与画面显示。

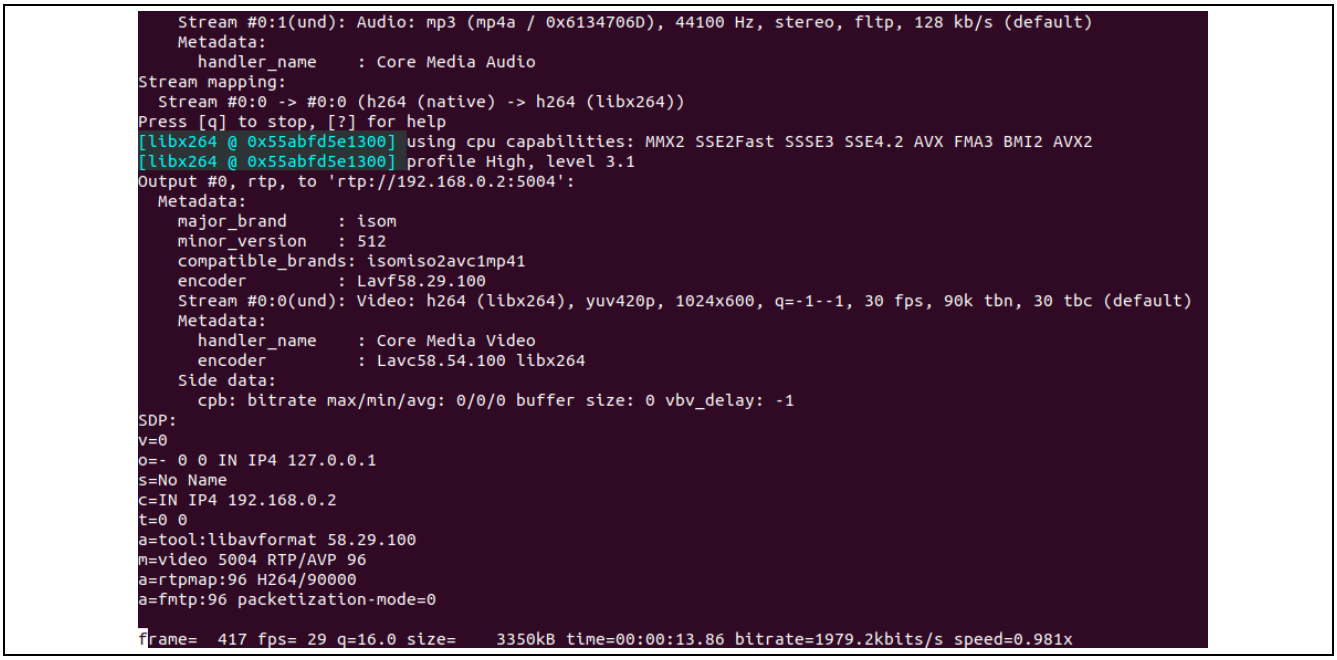


图 5-1 测试指令画面

6. 修订纪录

Date	Revision	Description
2026.05.11	1.00	初始发布。

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.