在 MA35D1 上實現即時串流影片

NuMicro® 64 位元系列微控制器範例程式碼介紹

文件資訊

應用簡述	在 MA35D1 FreeRTOS 平台上，使用 Real-timeTransport Protocol 傳輸串流影片，並利用 MA35D1 上的 H.264 decoder 解碼並播放。
BSP 版本	MA35D1_NonOS_BSP V1.4
開發平台	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

本範例程式展示如何在 Nuvoton MA35D1 FreeRTOS 平台上，結合 lwIP 網路協定棧與 RTP (Real-time Transport Protocol) 技術，實現高效能的串流影片即時解碼與顯示系統。與傳統使用 RTSP 協定不同，本方案採用更為輕量直接的 UDP Server 接收方式，跳過信令協商流程，直接處理來自 IPCAM 的 RTP 封包，這對於追求極低延遲的嵌入式影像監控應用至關重要。

在實作機制上，系統啟動 FreeRTOS 後會立即初始化 MA35D1 內建的 H.264 硬體解碼器。為了確保網路封包在高位元率傳輸下的穩定性，本範例特別導入了 Ring Buffer (環形緩衝區) 處理機制：從網路層接收到的 RTP 資料會先暫存於 Buffer 中，經由程式解包、重組後，直接送入硬體解碼引擎進行處理並輸出至顯示介面。這種設計充分發揮了 MA35D1 的硬體加速能力，即便在 RTOS 環境下，也能提供流暢且穩定的影像播放體驗。

此範例程式是基於 MA35D1_NonOS_BSP V1.4 開發環境（https://github.com/OpenNuvoton/MA35D1_NonOS_BSP）進行開發。在開始執行前，請確保你已建置好對應的開發環境，並獲取相關驅動程式庫。

雖然本文是以 NuMaker-HMI-MA35D1-S1 V1.0 開發板作為實作示範，但其內容與核心程式碼適用於所有具備 H.264 硬體解碼功能的 MA35D1 系列 MPU。對於開發工業監控終端、智慧安防顯示器或任何需要高效影像串流處理的嵌入式產品，均具有參考價值。

1.1 原理

本系統旨在於 FreeRTOS 即時作業系統環境下，針對嵌入式影像監控與智慧顯示終端，提供極低延遲的 H.264 影像串流解決方案。系統核心整合了 lwIP 網路協定棧與 MA35D1 內建的硬體解碼引擎，其主要運作原理可分為以下幾個技術維度：

1.1.1 輕量化低延遲傳輸：基於 UDP 的 RTP 直連解析

與傳統依賴 RTSP (Real-Time Streaming Protocol) 的串流方案不同，本系統在架構上進行了大幅度的輕量化。

- 免除協議交握開銷：傳統 RTSP 需經歷複雜的連線交握與狀態控制（如 DESCRIBE、SETUP、PLAY 等），容易造成影像顯示的首幀延遲 (First Frame Out Delay)。本方案直接部署 UDP Server 來接收 RTP (Real-time Transport Protocol) 封包，跳過所有前端信令協商。

- 即時封包處理：來自 IPCAM 的視訊資料透過 RTP 封包形式抵達後，系統會直接透過 lwIP 擷取 UDP 負載（Payload），並解析 RTP 標頭以獲取序列號與時間戳記，確保即時影像的時序正確性。此策略極大地縮短了網路端到應用端的資料傳遞路徑。

1.1.2 穩定資料流控：Ring Buffer 環形緩衝機制

在工業應用與高畫質監控場景中，高位元率（High Bitrate）的 H.264 串流容易因瞬間網路波動或 RTOS 任務切換產生突發資料量。為確保解碼穩定性，系統導入了 Ring Buffer 機制。

- 非同步生產與消費：網路接收任務（Producer）將解包後的 RTP 視訊負載快速寫入 Ring Buffer；而解碼任務（Consumer）則依據硬體引擎的處理節奏，從 Buffer 中讀取完整的 H.264 影格（Frame）資料。
- 防抖動與平滑化：透過此環形緩衝區，系統能有效吸收網路傳輸過程中的 Jitter（延遲抖動），避免硬體解碼器因資料匱乏（Underflow）而產生畫面卡頓，或因資料溢位（Overflow）導致丟幀。

1.1.3 高效能影像處理：MA35D1 核心硬體解碼加速

在解開 RTP 封包並重組出標準的 H.264 網路抽象層單元 (NALU, Network Abstraction Layer Unit) 後，資料會直接交由 MA35D1 的硬體解碼器進行處理。

- 卸載 CPU 運算壓力：將繁重的視訊解碼工作交由專屬硬體引擎執行，釋放了 Cortex-A35 核心的運算資源，使其能專注於 FreeRTOS 的即時任務調度、網路封包收發及其他 HMI 控制邏輯。
- 直通顯示介面：硬體解碼器輸出 YUV 或 RGB 原始影像資料後，可無縫銜接 MA35D1 的顯示控制器（Display Controller），直接輸出至 LCD 面板，完成從網路端到視覺終端的高效資料流轉。

1.1.4 整體資料流流程 (Data Flow Pipeline)

綜上所述，系統的標準處理管線（Pipeline）如下：

- 接收： lwIP 於 FreeRTOS 中監聽 UDP 端口，接收來自 IPCAM 的 RTP 封包。
- 解包與緩衝：系統去除 UDP/RTP 標頭，提取 H.264 NALU 資料並依序推入 Ring Buffer。

- 硬體解碼：解碼任務自 Ring Buffer 提取完整影格，驅動 MA35D1 H.264 硬體解碼引擎進行即時解碼。
- 渲染輸出：解碼後的高幀率影像直接推送至顯示介面完成畫面更新。

1.2 環境設定

本範例程式以 C 語言撰寫，並運行於 FreeRTOS 即時作業系統環境中，執行過程中會高度依賴底層硬體暫存器設定與 lwIP 網路協定棧。因此，在開始編譯與執行前，必須先確認相關的整合開發環境 (IDE)、軟體開發套件 (BSP) 以及燒錄工具均已正確安裝並設定完畢。

由於 Nuvoton 官方提供的工具鏈已針對 MA35D1 平台進行了完善的整合，本說明將以原廠標準工具作為開發與執行環境。您需要準備並確認以下開發工具：

- 整合開發環境 (IDE) - NuEclipse （V1.02.029）：

  本範例的專案建置、程式編譯與除錯均使用 NuEclipse 進行。關於 IDE 詳細安裝步驟與操作指南，請參考 NuEclipse User Manual。

- 軟體開發套件 (BSP) - MA35D1 Non-OS BSP：

  系統底層的 H.264 驅動與網路控制器操作皆基於官方 Non-OS BSP (包含 FreeRTOS 核心)。請確保您已自 GitHub 專案頁面 獲取對應版本。關於 BSP 的架構說明與 API 使用方式，可參考 NuMicro® Family MA35D1 Non-OS BSP User Manual。

- 映像檔燒錄工具 - NuWriter （V1.08）：

  當程式碼編譯完成並產出執行檔 (Image) 後，需透過 NuWriter 工具，將映像檔燒錄至開發板的儲存媒體中（如 SD Card、NAND Flash 或 eMMC）或 DDR 執行模式以供開機執行。

1.2.1 匯入 NuEclipse 專案

開啟 NuEclipse，再依下列步驟匯入本範例：

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

匯入完成後，對專案點選右鍵後選擇 Build Project。

1.3 燒錄與執行結果

在開始編譯之前，請前往以下路徑：

MA35D1_NonOS_BSP/Library/VC8000Lib

請將目錄下的 libvc8000_RTOS.a 重新命名，並覆蓋原有的 libvc8000.a 檔案。此檔案是專為 FreeRTOS 環境進行優化的驅動程式函式庫。

編譯完成後，產生的執行檔 (.bin) 將會輸出於以下路徑：

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

本範例將使用 NuWriter 工具將程式下載至 DDR 中直接執行。請依照以下步驟進行操作：

1. 連接硬體介面：請確保開發板已正確連接必要的線路，包含除錯埠 (Debug port，用於終端機訊息輸出)、USB 裝置端 (USBD，供 NuWriter 連線下載使用)，以及乙太網路 (Ethernet，用於接收 RTP 影像串流)。

2. 開啟開發模式：啟動 NuWriter 後，點選上方選單的 Mode -> Develop mode 以開啟開發模式。

3. 連接開發板：切換至 Attach 頁面，於下拉選單中選擇對應開發板型號的 Part Number，接著點擊 Attach 按鈕建立連線。

4. 設定下載參數：連線成功後（參考圖 1-1），切換至 Download 頁面，並選擇 DDR/SRAM 標籤。

5. 載入並執行：點選瀏覽並選擇上述編譯完成的執行檔位置，確保執行位址（Execute Address）設定為 0x80400000，最後點擊 Write 完成程式下載與執行。

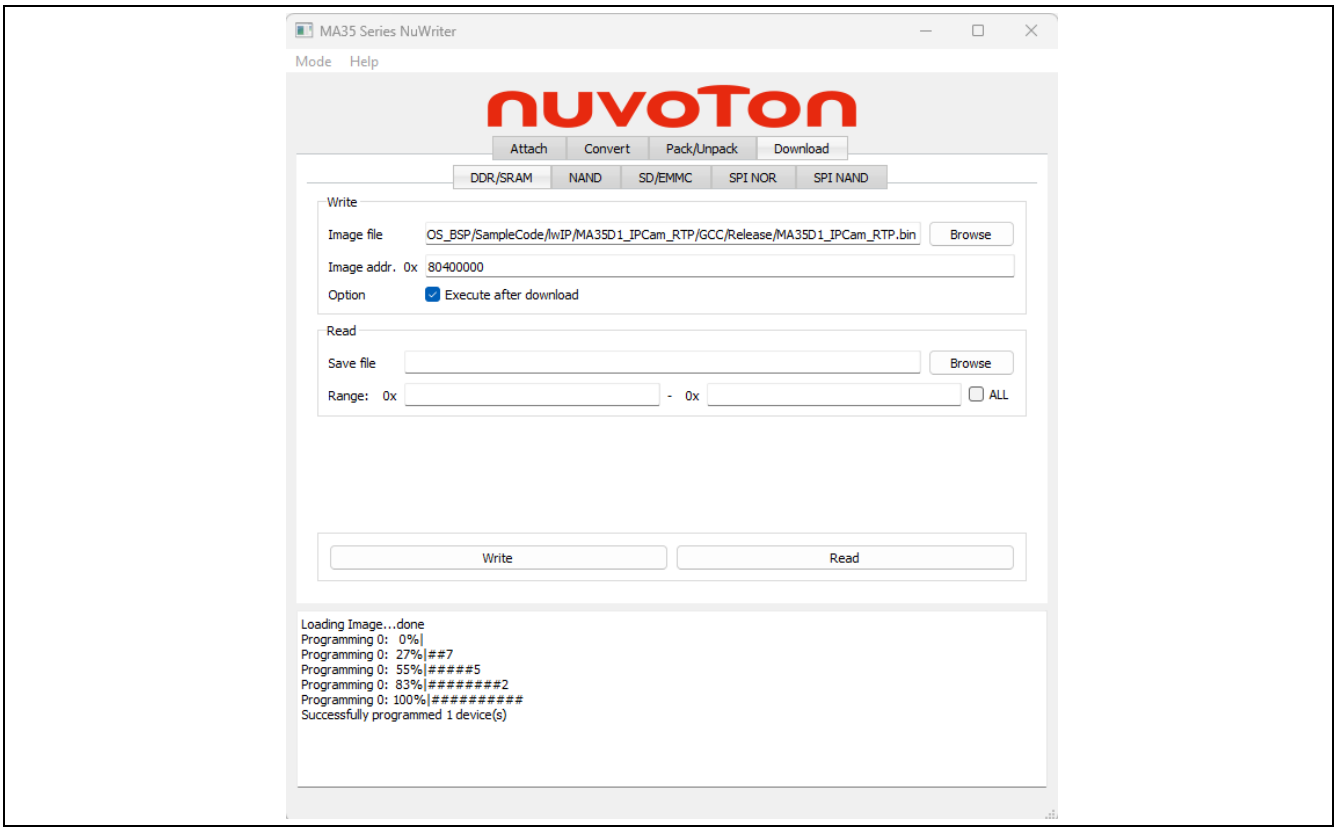


圖 1-1 NuWriter 燒錄畫面

燒錄完成後，可以從終端機看到如圖 1-2 的除錯的訊息，後續即可開始接收網路封包解碼並播放影片。

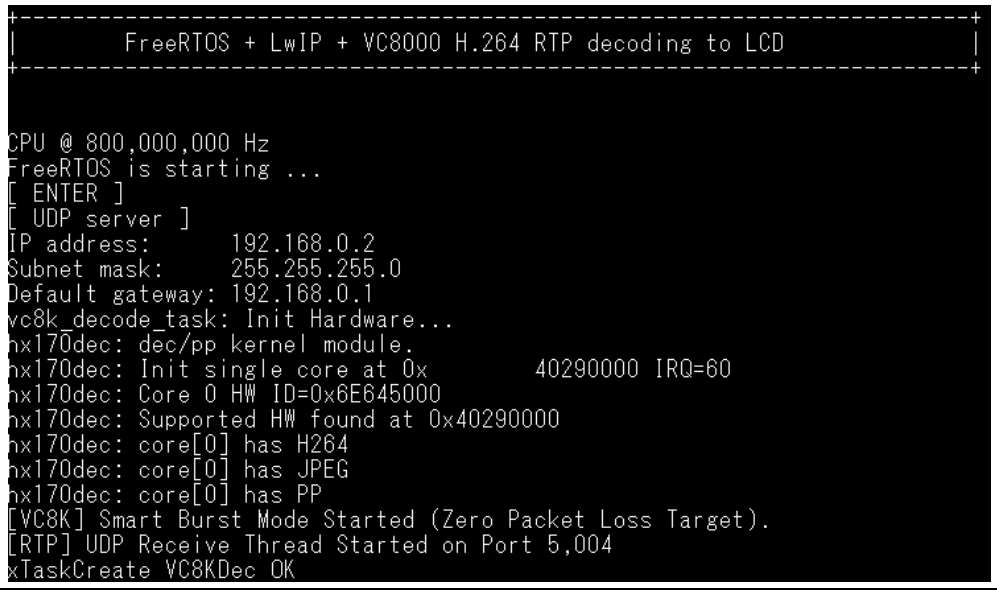


圖 1-2 終端機畫面

2. 程式碼介紹

2.1 main.c

這支 main.c 程式是整個專案的系統總指揮 (Entry Point)。它的主要任務是在 Nuvoton MA35D1 微處理器上，建立一個基於 FreeRTOS 的「即時影像串流接收與播放系統」。

具體來說，它負責嚴密地協調以下三大硬體與軟體模組的運作：

- 視覺輸出 (Display)：在系統開機時，初始化系統時脈與 LCD 顯示控制器，並規劃好 Frame Buffer，確保螢幕能正常顯示開機 Logo 與後續的動態影像。
- 網路接收 (Network)：啟動 lwIP 網路協定棧，設定開發板的 IP 位址，並建立一個 UDP 伺服器。這個伺服器專門用來接收來自遠端 (如 IP Camera) 的 RTP (H.264) 視訊封包，並將其推入 Ring Buffer 緩衝區。
- 硬體解碼 (Decoding)：配置好專屬的記憶體空間後，喚醒 VC8000 硬體解碼任務。該任務會不斷從緩衝區提取影像資料，交由硬體加速解碼，最後直接輸出至螢幕。

2.1.1 系統硬體記憶體佈局 (Memory Layout)

在處理高畫質影像解碼時，精確的記憶體配置至關重要。由於 MA35D16A887C 具備 256 MB 的 DDR 記憶體，本程式碼開頭即明確定義了各個硬體加速單元專屬的實體記憶體位址。系統將 0x80000000 前段留給程式碼與 FreeRTOS 堆疊使用，並將 0x88000000 之後的連續大空間嚴格劃分給 LCD Frame Buffer (顯示緩衝)、RTP Ring Buffer (1 MB 網路接收緩衝) 以及 VC8000 硬體解碼器專屬的 Heap (32 MB)。這種直接指定實體位址的靜態劃分方式，能有效避免 RTOS 執行期間因頻繁配置大記憶體而產生的碎片化或溢位風險。

```
/* ---------------- 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 網路與解碼任務橋接配置 (vUdpTask)

vUdpTask 是串接「網路接收」與「硬體解碼」兩大系統核心的關鍵任務。程式首先透過 lwIP 協定棧設定開發板的靜態 IP 位址 (192.168.0.2) 並啟用網路介面。接著，將 RTP Ring Buffer 初始化綁定至預先規劃的 DDR 位址上，並啟動負責接收影像封包的 UDP Server。最後，透過 xTaskCreate 建立 vc8k_decode_task 任務，確立了由 UDP 網路端作為「生產者」，VC8000 硬體解碼端作為「消費者」的高效非同步架構。

```
/* 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 主程式系統與顯示介面初始化 (main)

main 函式為系統啟動的總控制中心。在完成底層時鐘與硬體初始化 (SYS_Init) 後，程式會先進行開機畫面的處理：透過指標運算計算開機 Logo 的大小，並利用 memcpy 將影像資料直接搬移至 DDR 中的 Frame Buffer 位址。隨後呼叫 DISPLIB_SetFBConfig，將 LCD 顯示控制器精準導向該 Frame Buffer 位址並啟用視訊層輸出。一切硬體就緒後，便建立 UdpTask 任務並正式啟動 FreeRTOS 系統排程器，讓系統從單一執行緒進入強大的多工作業階段。

```
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

這份 rtp_h264.c 的核心功能是將網路層接收到的 RTP 封包，轉換為硬體解碼器可識別的 H.264 Annex-B 串流格式（即帶有 00 00 00 01 啟始碼的 NALU）。

2.2.1 主解析入口與 NAL 分流機制 (h264_rtp_to_annexb)

h264_rtp_to_annexb 為此模組的主要入口函數。它首先會嚴格驗證 RTP 標頭（Header）的合法性，並處理 Extension 與 Padding 欄位，確保精準定位到 H.264 Payload。特別重要的是，此處加入了防禦性設計：主動丟棄對硬體解碼無用的 SEI (Type 6) 與 AUD (Type 9) 封包，藉此保護後端解析器並節省記憶體頻寬。最後再根據 NAL 類型，將資料分流給對應的處理函數。

```
/* 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 碎片重組與防錯機制 (handle_fu_a)

當 H.264 影格（如 I-Frame）過大超過網路 MTU 時，會被切分成多個 RTP 封包，稱為 FU-A。此函數透過解析 S (Start) 與 E (End) 位元來還原資料。遇到 Start 時，會重建原始的 NAL Header 並加上 Annex-B 啟始碼。為因應 UDP 掉包風險，程式使用了 s_in_fu_a 全域變數進行狀態追蹤；若遺失了 Start 封包，後續的碎片（孤兒碎片）將被安全地丟棄，避免將殘缺的畫面送入解碼器導致系統崩潰或畫面破圖。

```
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

本程式碼 (rtp_receiver.c) 在整個即時串流系統中扮演「資料生產者」的角色，專責處理底層網路收發。它利用 lwIP 的 Netconn API 建立了一個獨立的 FreeRTOS 網路接收任務，持續監聽指定的 UDP 埠口 (預設為 5004)。當接收到來自遠端 (如 IP Camera) 的 RTP 影像封包時，系統會將封包內容擷取出來，並交由後端的 H.264 解析模組進行處理，是串接實體網路與影像解碼引擎的第一道關鍵橋樑。

2.3.1 網路接收任務 (rtp_udp_recv_thread)

此函數是負責實際接收網路封包的核心執行緒。為了滿足高畫質影像頻繁收發的嚴苛要求並確保系統長時運作的穩定性，此函數具備以下幾個關鍵設計：

- 靜態記憶體配置：特別宣告了固定大小的靜態陣列 (payload_buf) 作為接收緩衝區，徹底避開了因頻繁呼叫 malloc/free 所引發的效能損耗與記憶體碎片化 (Memory Fragmentation) 風險。
- 阻塞式等待與安全防護：執行緒以阻塞模式 (netconn_recv) 停駐等待封包，避免無謂的 CPU 資源消耗。收到封包後，會優先進行長度檢查 (len > RTP_MAX_UDP_PAYLOAD)，主動丟棄過大的異常封包以防範緩衝區溢位。
- 資源釋放防漏：在完成封包的複製與解析調用後，程式必定會執行 netbuf_delete(buf) 來釋放 lwIP 內部的資料鏈結區塊，這是防止系統發生記憶體洩漏 (Memory Leak) 的關鍵步驟。

```
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

本程式碼實作針對硬體影像解碼最佳化之進階環形緩衝區 (Ring Buffer)。傳統環形緩衝區在寫入指標到達邊界時會產生折返 (Wrap-around) 現象，導致單一 H.264 影格資料在記憶體位址上不連續。由於 MA35D1 內建之 VC8000 硬體解碼引擎依賴 DMA 進行資料存取，其要求實體記憶體位址必須絕對連續。

為解決位址不連續之限制並最大化系統效能，本模組導入「隱藏備用區 (Hidden Headroom Backup)」機制。透過在緩衝區前端保留特定大小之記憶體區塊，當發生資料折返時，系統能

快速將破碎資料拼接為連續區塊，實現零拷貝線性化 (Zero-Copy Linearization)，有效免除額外配置暫存區與整包資料拷貝的龐大系統開銷。

2.4.1 零拷貝線性化 API (rtp_ring_peek_linear)

此函數為本模組之核心，其功能為向硬體解碼任務提供「實體位址絕對連續」之影像資料指標。其底層運作機制分為以下兩種情境：

- 情境一（實體位址連續）：

  若當前寫入指標 (wr) 位於讀取指標 (rd) 之後，表示資料未跨越緩衝區邊界，具備實體連續性。此時系統不進行任何記憶體搬移，直接回傳當前位址與可讀長度，無額外 CPU 運算開銷。

- 情境二（折返線性拼接）：

  若資料發生折返，導致完整影格被分割為「尾段 (Tail)」與「頭段 (Head)」。系統會將尾段資料複製至預先保留之 128 KB 隱藏備用區，使其與頭段資料在實體位址上完美銜接。同時，系統重定向讀取指標至備用區，將原本破碎的資料轉化為單一連續指標供解碼器讀取。

```
/**
 * @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

本程式碼 (vc8k_decode.c) 實作了驅動 MA35D1 內建 VC8000 硬體解碼引擎的 FreeRTOS 獨立任務，在系統中扮演「資料消費者」的角色。其主要職責是從網路接收任務寫入的環形緩衝區 (Ring Buffer) 中提取 H.264 視訊串流，並將其送入硬體引擎進行高效解碼。解碼完成後，透過硬體內建的後處理器 (Post-Processor, PP)，系統會將影像直接轉換為 LCD 面板所需的色彩格式並輸出至顯示緩衝區 (Frame Buffer)。

2.5.1 硬體解碼任務與後處理初始化 (vc8k_decode_task)

此函數為解碼執行緒的進入點，負責完成 VC8000 核心的硬體初始化與專屬記憶體 (Heap) 配置。其中最具技術價值的設計在於「硬體後處理器 (Post-Processor)」的配置：程式會依據 LCD 面板的解析度，直接將硬體解碼後的輸出格式設定為 VC8000_PP_F_RGB888，並將目標實體位址 (pp_out_paddr) 綁定至系統保留的 Frame Buffer 位址 (VC8K_FB_ADDR)。這項設定確保了影像一旦解碼完畢，硬體便會自動完成 YUV 到 RGB 的色彩空間轉換並寫入顯示記憶體。

```
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 智慧解碼迴圈與快取同步機制 (VC8000_BitstreamDecodeLoop)

此函數是持續將資料送入硬體解碼引擎的核心無限迴圈。為了確保 RTOS 效能與硬體穩定性，本函數導入了幾個關鍵機制：

1. 智慧爆發模式 (Smart Burst)：當緩衝區累積的資料量超過閾值 (MIN_BURST_SIZE = 4 KB) 時才喚醒硬體處理，有效避免因處理微小封包而產生頻繁的 RTOS 上下文切換 (Context Switch) 開銷。

2. 快取一致性管理 (Cache Flush)：由於 VC8000 是透過 DMA 直接讀取外部記憶體，程式在呼叫硬體解碼前，必定會執行 dcache_clean_by_mva。此動作能將 CPU 暫存在 Cache 中的影像資料強制刷新至實體 DDR 中，防止硬體讀取到未更新的舊資料而導致畫面破圖。

3. 動態緩衝區更新：硬體執行完畢後，會回傳實際未消耗的位元組數 (remain)，程式依此精準計算出硬體實際吃掉的資料量 (consumed)，並同步更新 Ring Buffer 的讀取指標，維持系統資料流的穩定平衡。

```
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. 軟體與硬體需求

3.1 軟體需求

- 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 硬體需求

- 電路元件
  - NuMaker-HMI-MA35D1-S1 V1.0

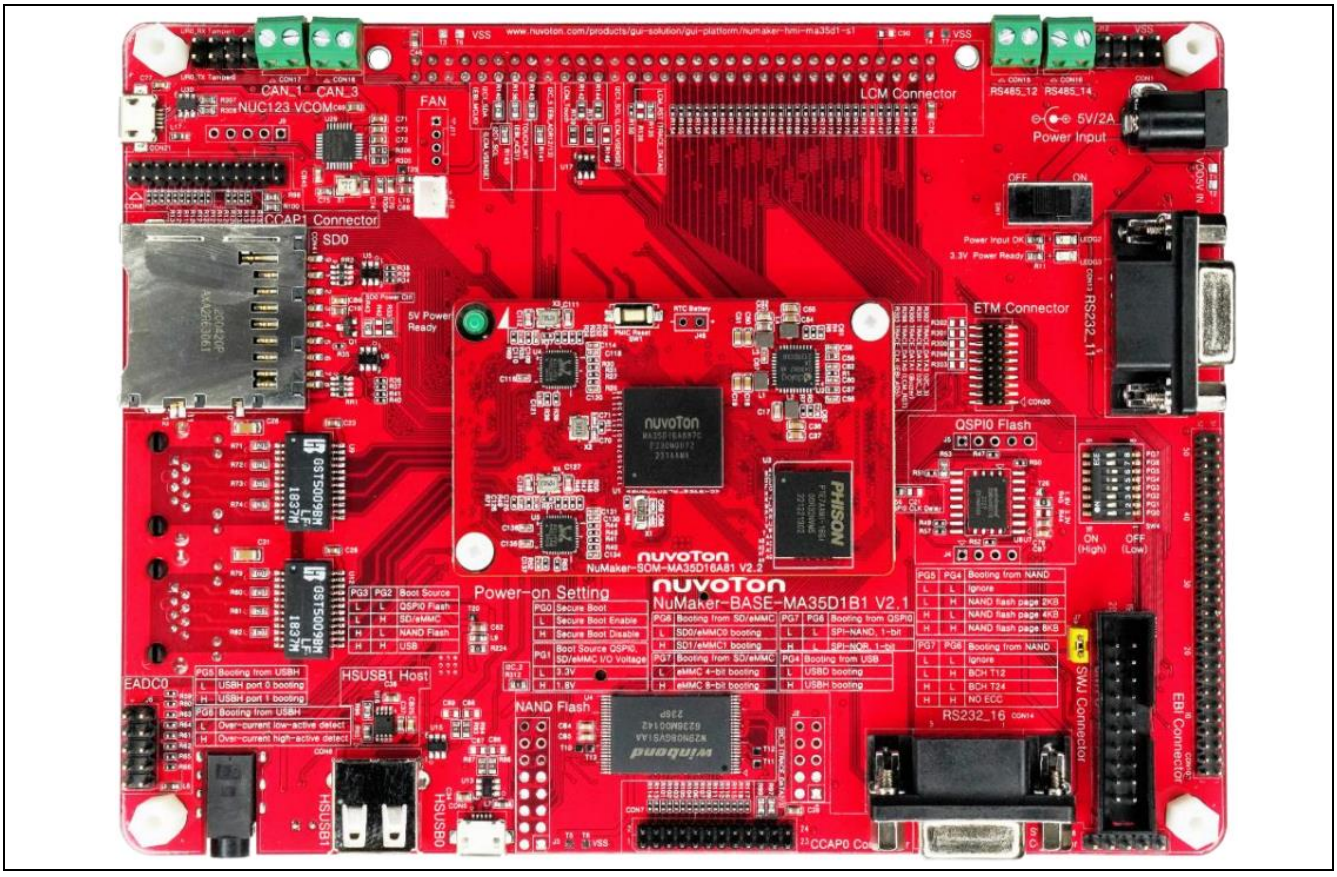


圖 3-1 NuMaker-HMI-MA35D1-S1 V1.0

4. 目錄資訊

目錄結構如下圖所示

EC_MA35D1_IPCam_RTP_V1.00	
Library	Driver source file
Arch	Architecture-specific source file
DisplayLib	Display source file
StdDriver	Standard driver of MA35D1 series
VC8000Lib	VC8000 video decoder source file
SampleCode	Nuvoton ARM64 device tree files
lwIP	
port	lwIP source file
MA35D1_IPCam_RTP	Source file of sample cod
ThirdParty	Third party source file

圖 4-1 目錄資訊

5. 範例程式執行

本範例程式於如 1.3 章節燒錄完成後，會開始監聽並等待接收 H.264 RTP 串流封包。要將視訊封包發送至開發板的 IP 位址，您可以使用實體的 IP Camera，或是在 PC 端透過指令模擬發送。

以下說明如何利用 PC (以 Ubuntu 系統為例) 上的 FFmpeg（version 4.2.7-0ubuntu0.1）工具來模擬 IPCAM 發送 RTP 封包：

1. 準備測試影片：請準備一段 H.264 格式的測試影片（本範例使用 1024600.mp4），並將該影片檔放置於 Ubuntu 系統準備執行指令的當下資料夾路徑中。

2. 執行 FFmpeg 推流指令：開啟 Ubuntu 終端機 (Terminal)，輸入以下指令以原生幀率讀取影片，並透過 RTP 協定發送至開發板。

```
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. 確認推流狀態：指令執行後，Ubuntu 終端機將會持續印出目前的轉播進度與傳輸速率，PC 端的畫面請參考圖 5-1。此時，開發板端應已成功接收 RTP 封包，並開始進行硬體解碼與畫面顯示。

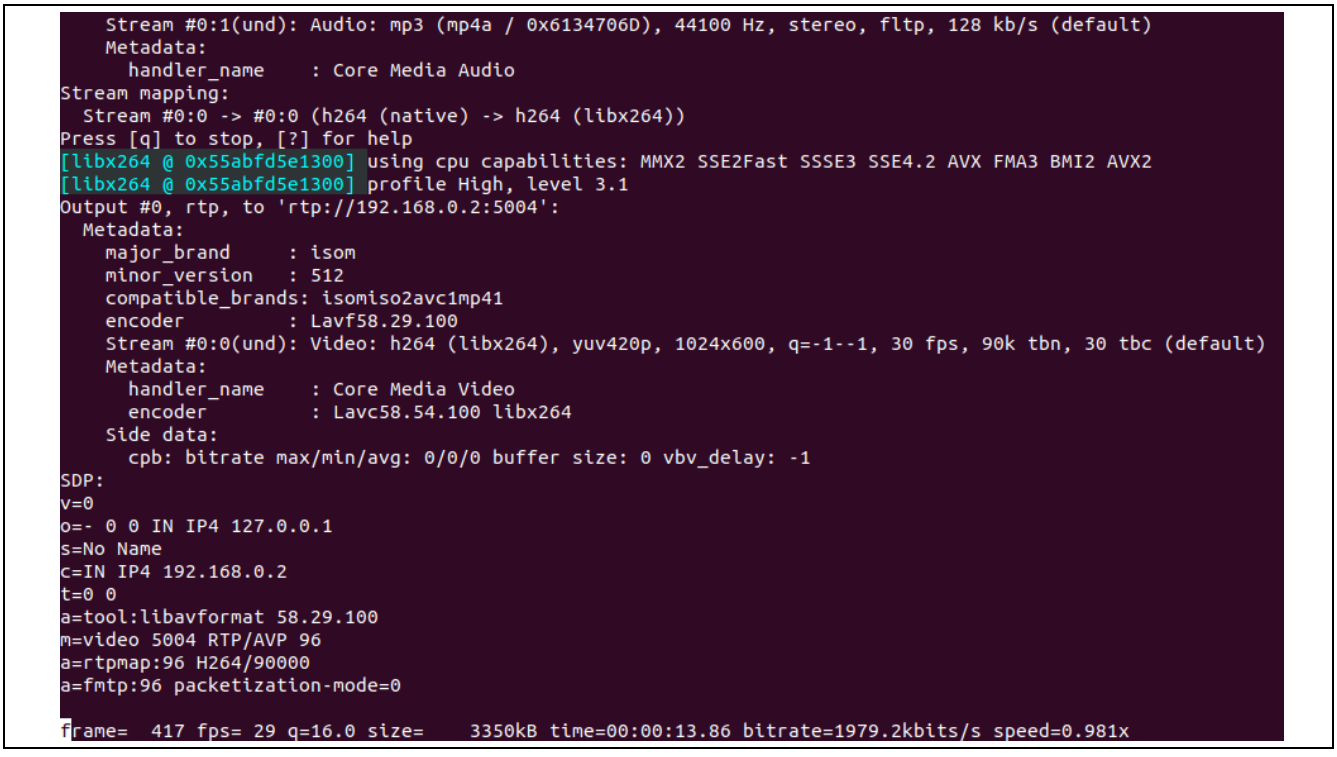


圖 5-1 測試指令畫面

6. 修訂紀錄

Date	Revision	Description
2026.05.11	1.00	初始發布。

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.