Real-Time Video Streaming on MA35D1

Example Code Introduction for 64-bit NuMicro® Family

Document Information

Application	Implementing video streaming on the MA35D1 FreeRTOS platform using the Real-time Transport Protocol (RTP) for transmission, and leveraging the MA35D1's built-in H.264 decoder for decoding and playback.
BSP Version	MA35D1_NonOS_BSP V1.4
Hardware	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. Overview

This sample program demonstrates how to implement a high-performance, real-time video streaming decoding and display system on the Nuvoton MA35D1 FreeRTOS platform by integrating the lwIP network stack with RTP (Real-time Transport Protocol) technology. Unlike traditional approaches that rely on the RTSP protocol, this solution adopts a more lightweight and direct UDP Server reception method. By bypassing the signaling negotiation process and directly processing RTP packets from IP cameras, this architecture is crucial for embedded video surveillance applications that demand ultra-low latency.

In terms of implementation, the system immediately initializes the MA35D1's built-in H.264 hardware decoder upon starting FreeRTOS. To ensure the stability of network packets during high-bitrate transmissions, this example specifically introduces a Ring Buffer mechanism. RTP data received from the network layer is temporarily stored in the buffer, unpacked, and reconstructed by the software before being sent directly to the hardware decoding engine for processing and outputting to the display interface. This design fully leverages the hardware acceleration capabilities of the MA35D1, delivering a smooth and stable video playback experience even within an RTOS environment.

This sample program is developed based on the MA35D1_NonOS_BSP V1.4 development environment (https://github.com/OpenNuvoton/MA35D1_NonOS_BSP). Before execution, please ensure that you have correctly set up the corresponding development environment and acquired the necessary driver libraries.

Although this document uses the NuMaker-HMI-MA35D1-S1 V1.0 development board for the implementation demonstration, its concepts and core code are applicable to all MA35D1 series MPUs equipped with H.264 hardware decoding capabilities. It serves as an ideal reference for developing industrial monitoring terminals, smart security displays, or any embedded products requiring highly efficient video stream processing.

1.1 Principle

Designed for the FreeRTOS real-time operating system environment, this system provides an ultra-low latency H.264 video streaming solution tailored for embedded video surveillance and smart display terminals. The system core integrates the lwIP network stack with the MA35D1's built-in hardware decoding engine. Its primary operational principles can be categorized into the following technical aspects:

1.1.1 Lightweight & Low-Latency Transmission: UDP-Based Direct RTP Parsing

Unlike traditional streaming solutions that rely on the RTSP (Real-Time Streaming Protocol), this system features a significantly streamlined architecture.

- Eliminating Protocol Handshake Overhead: Traditional RTSP requires complex connection handshakes and state controls (e.g., DESCRIBE, SETUP, PLAY), which often cause First Frame Out Delays. This solution directly deploys a UDP Server to receive RTP (Real-time Transport Protocol) packets, bypassing all front-end signaling negotiations.

- Real-Time Packet Processing: Once video data from the IPCAM arrives via RTP packets, the system directly extracts the UDP payload through lwIP and parses the RTP header to retrieve the sequence number and timestamp, ensuring accurate timing for the real-time video. This strategy drastically shortens the data transmission path from the network layer to the application layer.

1.1.2 Stable Data Flow Control: Ring Buffer Mechanism

In industrial applications and high-definition surveillance scenarios, high-bitrate H.264 streams are prone to data bursts caused by momentary network fluctuations or RTOS task switching. To ensure decoding stability, the system incorporates a Ring Buffer mechanism.

- Asynchronous Production and Consumption: The network receiving task (Producer) rapidly writes the unpacked RTP video payload into the Ring Buffer, while the decoding task (Consumer) reads complete H.264 frame data from the buffer according to the processing pace of the hardware engine.
- Anti-Jitter and Smoothing: Utilizing this circular buffer, the system effectively absorbs network transmission jitter, preventing the hardware decoder from experiencing screen stuttering due to data starvation (Underflow) or dropped frames due to data overflow.

1.1.3 High-Performance Video Processing: MA35D1 Hardware Decoding Acceleration

After unpacking the RTP packets and reconstructing standard H.264 Network Abstraction Layer Units (NALUs), the data is directly passed to the MA35D1 hardware decoder for processing.

- Offloading CPU Computation: Delegating heavy video decoding workloads to the dedicated hardware engine frees up the Cortex-A35 core's computing resources. This allows the core to focus exclusively on FreeRTOS real-time task scheduling, network packet handling, and other HMI control logic.
- Direct Display Interface: Once the hardware decoder outputs raw YUV or RGB image data, it interfaces seamlessly with the MA35D1 Display Controller. The data is directly output to the LCD panel, completing a highly efficient data flow pipeline from the network layer to the visual terminal.

1.1.4 Overall Data Flow Pipeline

In summary, the standard processing pipeline of the system operates as follows:

- Reception: lwIP listens to the UDP port within FreeRTOS, receiving RTP packets from the IPCAM.
- Unpacking & Buffering: The system strips the UDP/RTP headers, extracts the H.264 NALU data, and sequentially pushes it into the Ring Buffer.
- Hardware Decoding: The decoding task pulls complete frames from the Ring Buffer and drives the MA35D1 H.264 hardware engine to perform real-time decoding.
- Rendering & Output: The decoded high-framerate video is pushed directly to the display interface to complete the screen update.

1.2 Environment Setup

This sample program is written in C and runs within the FreeRTOS real-time operating system environment. During execution, it heavily relies on underlying hardware register configurations and the lwIP network stack. Therefore, before compiling and executing the code, you must ensure that the Integrated Development Environment (IDE), Board Support Package (BSP), and flashing tool are all correctly installed and configured.

Because the official toolchain provided by Nuvoton is thoroughly integrated for the MA35D1 platform, this guide uses the official standard tools for the development and execution environment. Please prepare and verify the following development tools:

- Integrated Development Environment (IDE) - NuEclipse (V1.02.029):

  Project building, code compilation, and debugging for this example are all performed using NuEclipse. For detailed installation steps and operating instructions, please refer to the NuEclipse User Manual.

- Board Support Package (BSP) - MA35D1 Non-OS BSP:

  The system's underlying H.264 driver and network controller operations are based on the official Non-OS BSP (which includes the FreeRTOS core). Please ensure that you have acquired the corresponding version from the GitHub project page. For an overview of the BSP architecture and API usage, please refer to the NuMicro® Family MA35D1 Non-OS BSP User Manual.

- Image Flashing Tool - NuWriter (V1.08):

  Once the code is compiled and the executable file (Image) is generated, you will need the NuWriter tool to flash the image into the development board's storage media (such as an SD Card, NAND Flash, or eMMC) or load it into DDR mode for direct execution upon boot.

1.2.1 Importing the NuEclipse Project

Open NuEclipse, and follow the steps below to import the project:

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

Once imported, right-click on the project and select Build Project to start the compilation process.

1.3 Flashing and Execution Results

Pre-compilation Step: Replace the Optimized Library

Before compiling, please navigate to the following path:

MA35D1_NonOS_BSP/Library/VC8000Lib

Rename the libvc8000_RTOS.a file to libvc8000.a and replace the original file. This specific library is a driver version optimized for the FreeRTOS environment.

After compilation is complete, the generated executable file (.bin) will be output to the following path:

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

This example uses the NuWriter tool to download the program directly into the DDR for execution. Please follow the steps below:

1. Connect the Hardware Interfaces: Ensure the development board is properly connected with the necessary cables, including the Debug port (for terminal message output), USB Device (USBD, for NuWriter connection and downloading), and Ethernet (for receiving the RTP video stream).
2. Enable Develop Mode: Launch NuWriter, then click Mode -> Develop mode from the top menu to enable the development mode.
3. Connect to the Development Board: Switch to the Attach page, select the Part Number corresponding to your development board model from the drop-down menu, and then click the Attach button to establish a connection.
4. Configure Download Parameters: Upon a successful connection (refer to Figure 1-1), switch to the Download page and select the DDR/SRAM tab.
5. Load and Execute: Click browse to select the compiled executable file from the path mentioned above. Ensure that the Execute Address is set to 0x80400000, and finally, click Write to complete the download and start execution.

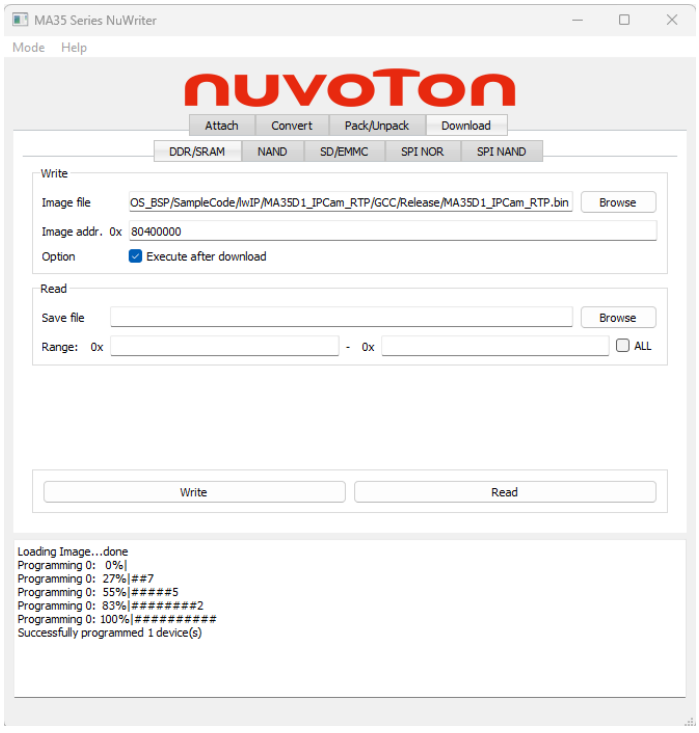


Figure 1-1 NuWriter Programming Screen

Once the download is complete, you can see the debug messages in the terminal as shown in Figure 1-2. After that, the system will be ready to receive network packets, decode them, and play the video.

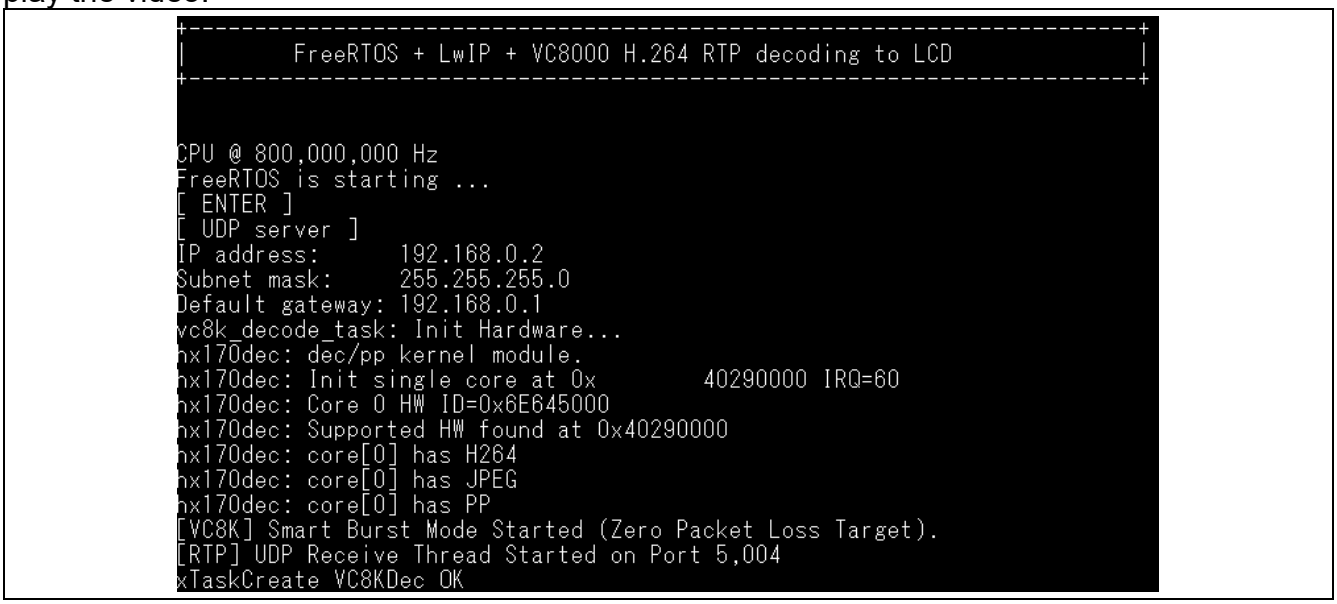


Figure 1-2 Terminal Screen

2. Code Description

2.1 main.c

This main.c program serves as the system controller (Entry Point) for the entire project. Its primary objective is to establish a FreeRTOS-based "real-time video streaming reception and playback system" on the Nuvoton MA35D1 microprocessor.

Specifically, it is responsible for strictly coordinating the operations of the following three major hardware and software modules:

- Visual Output (Display): Upon system boot, it initializes the system clocks and the LCD display controller, and configures the Frame Buffer. This ensures the screen can properly display the startup logo and subsequent dynamic video frames.
- Network Reception (Network): It initializes the lwIP network stack, configures the development board's IP address, and sets up a UDP server. This server is dedicated to receiving RTP (H.264) video packets from remote sources (such as an IP Camera) and pushing them into the Ring Buffer.
- Hardware Decoding (Decoding): After allocating the dedicated memory space, it launches the VC8000 hardware decoding task. This task continuously extracts video data from the buffer, offloads it to the hardware accelerator for decoding, and outputs the frames directly to the screen.

2.1.1 System Hardware Memory Layout

Precise memory allocation is crucial when handling high-definition video decoding. Because the MA35D16A887C is equipped with 256 MB of DDR memory, the beginning of the code explicitly defines the dedicated physical memory addresses for each hardware acceleration unit.

The system reserves the memory segment starting at 0x80000000 for the executable code and FreeRTOS stacks. It then strictly partitions the large, continuous memory space starting from 0x88000000 for the LCD Frame Buffer, the RTP Ring Buffer (a 1 MB network reception buffer), and the dedicated Heap for the VC8000 hardware decoder (32 MB). This static partitioning approach, which directly assigns physical addresses, effectively eliminates the risks of memory fragmentation and overflow that can occur from frequently allocating large memory blocks during RTOS execution.

```
/* ---------------- 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 Network and Decoding Task Bridge Configuration (vUdpTask)

The vUdpTask is the critical task that bridges the two core system modules: "Network Reception" and "Hardware Decoding." First, the program utilizes the lwIP protocol stack to configure the development board's static IP address (192.168.0.2) and enable the network interface. Next, it initializes and binds the RTP Ring Buffer to the pre-allocated DDR address, and starts the UDP Server responsible for receiving video packets. Finally, it uses xTaskCreate to create the vc8k_decode_task task, establishing a highly efficient asynchronous architecture where the UDP network end acts as the "Producer" and the VC8000 hardware decoding end acts as the "Consumer."

```
/* 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,            /* 16KB 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 Main System and Display Interface Initialization (main)

The main function serves as the central control hub for system startup. After completing the low-level clock and hardware initialization (SYS_Init), the program first processes the boot splash screen: it calculates the size of the boot logo via pointer arithmetic and utilizes memcpy to transfer the image data directly to the Frame Buffer address in the DDR memory. Subsequently, it calls DISPLIB_SetFBConfig to accurately map the LCD display controller to this Frame Buffer address and enable the video layer output. Once all hardware is fully initialized and ready, the program creates the UdpTask and officially starts the FreeRTOS system scheduler, transitioning the system from a single-threaded operation into a robust multitasking environment.

```
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

The core function of the rtp_h264.c file is to convert RTP packets received from the network layer into the H.264 Annex-B stream format recognized by the hardware decoder (i.e., NALUs prefixed with the 00 00 00 01 start code).

2.2.1 Main Parsing Entry Point and NAL Routing Mechanism (h264_rtp_to_annexb)

The h264_rtp_to_annexb acts as the primary entry function for this module. It first strictly validates the RTP header and processes the Extension and Padding fields to ensure the H.264 payload is accurately located. Crucially, a defensive design is implemented here: it proactively discards SEI (Type 6) and AUD (Type 9) packets, which are unnecessary for hardware decoding. This strategy protects the backend parser and conserves memory bandwidth. Finally, the data is routed to the corresponding processing functions based on the NAL unit type.

```
/* 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 Fragmentation Reassembly and Fault Tolerance Mechanism (handle_fu_a)

When an H.264 frame (such as an I-Frame) exceeds the network Maximum Transmission Unit (MTU), it is fragmented into multiple RTP packets, known as FU-A (Fragmentation Unit A). This function reassembles the data by parsing the S (Start) and E (End) bits. Upon detecting a Start bit, it reconstructs the original NAL Header and prepends the Annex-B start code.

To mitigate the inherent packet loss risks associated with UDP, the program utilizes the s_in_fu_a global variable for state tracking. If a Start packet is lost, subsequent fragments (orphan fragments) are safely discarded. This prevents incomplete frame data from being sent to the hardware decoder, thereby averting system crashes or visual artifacts (image corruption).

```
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

This source code (rtp_receiver.c) acts as the "Data Producer" within the entire real-time streaming system, specifically handling low-level network communication. It leverages lwIP's Netconn API to establish an independent FreeRTOS network reception task that continuously listens on a designated UDP port (default: 5004). Upon receiving RTP video packets from a remote source (such as an IP Camera), the system extracts the packet content and passes it to the backend H.264 parsing module, serving as the first critical bridge between the physical network and the video decoding engine.

2.3.1 Network Reception Task (rtp_udp_recv_thread)

This function is the core thread responsible for the actual reception of network packets. To meet the stringent requirements of frequent high-definition video transmission and to ensure long-term system stability, several key design features have been implemented:

- Static Memory Allocation: A fixed-size static array (payload_buf) is specifically declared as the reception buffer. This completely avoids the performance overhead and memory fragmentation risks associated with frequent malloc/free calls.
- Blocking Wait and Security Safeguards: The thread remains suspended in blocking mode (netconn_recv) while waiting for packets, preventing unnecessary CPU consumption. Once a packet arrives, the system performs a length check (len > RTP_MAX_UDP_PAYLOAD) and proactively discards oversized anomalous packets to prevent buffer overflows.
- Resource Management and Leak Prevention: After packet duplication and parsing are completed, the program must execute netbuf_delete(buf) to release internal lwIP data link blocks. This is a vital step in preventing system memory leaks.

```
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

This source code implements an advanced Ring Buffer optimized specifically for hardware video decoding. Traditional ring buffers experience a wrap-around phenomenon when the write pointer reaches the boundary, causing the data of a single H.264 frame to become non-contiguous in memory. Because the MA35D1's built-in VC8000 hardware decoding engine relies on DMA for data access, it requires physical memory addresses to be strictly contiguous.

To resolve the limitation of non-contiguous addresses and maximize system performance, this module introduces a "Hidden Headroom Backup" mechanism. By reserving a specifically sized memory block at the front of the buffer, the system can quickly splice fragmented data into a contiguous block when wrap-around occurs. This achieves Zero-Copy Linearization, effectively eliminating the massive system overhead associated with allocating additional temporary buffers and copying entire data packets.

2.4.1 Zero-Copy Linearization API (rtp_ring_peek_linear)

This function is the core of the module. Its purpose is to provide the hardware decoding task with a video data pointer that guarantees "strictly contiguous physical memory." Its underlying operational mechanism is divided into the following two scenarios:

- Scenario 1: Contiguous Physical Addresses

  If the current write pointer (wr) is located behind the read pointer (rd), it indicates that the data has not crossed the buffer boundary and maintains physical contiguity. In this case, the system does not perform any memory movement; it directly returns the current address and the readable length without incurring any additional CPU overhead.

- Scenario 2: Wrap-Around Linear Splicing

  If data wrap-around occurs, a complete video frame is split into a "Tail" section and a "Head" section. The system will copy the Tail data into the pre-reserved 128KB Hidden Headroom Backup area, allowing it to connect seamlessly with the Head data at the physical address level. Simultaneously, the system redirects the read pointer to this backup area, transforming the originally fragmented data into a single contiguous pointer for the decoder to read.

```
/**
 * @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

This source code (vc8k_decode.c) implements an independent FreeRTOS task that drives the MA35D1's built-in VC8000 hardware decoding engine, acting as the "Data Consumer" within the system. Its primary responsibility is to extract the H.264 video stream from the Ring Buffer (which is written to by the network reception task) and feed it into the hardware engine for highly efficient decoding. Once decoding is complete, the system utilizes the hardware's built-in Post-Processor (PP) to directly convert the video into the color format required by the LCD panel and output it to the Frame Buffer.

2.5.1 Hardware Decoding Task and Post-Processing Initialization (vc8k_decode_task)

This function serves as the entry point for the decoding thread. It is responsible for completing the hardware initialization of the VC8000 core and allocating its dedicated memory (Heap).

The most technically valuable design within this function lies in the configuration of the "Hardware Post-Processor". Based on the resolution of the LCD panel, the program directly sets the hardware-decoded output format to VC8000_PP_F_RGB888 and binds the target physical address (pp_out_paddr) to the system-reserved Frame Buffer address (VC8K_FB_ADDR). This configuration ensures that the moment a frame is decoded, the hardware automatically performs the YUV-to-RGB color space conversion and writes the data directly into the display memory.

```
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 Smart Decoding Loop and Cache Synchronization Mechanism (VC8000_BitstreamDecodeLoop)

This function is the core infinite loop that continuously feeds data into the hardware decoding engine. To ensure RTOS performance and hardware stability, this function incorporates several key mechanisms:

1. Smart Burst Mode: The hardware processing is only awakened when the accumulated data volume in the buffer exceeds a specific threshold (MIN_BURST_SIZE = 4KB). This effectively avoids the massive overhead of frequent RTOS context switches that would occur if the system processed tiny packets individually.

2. Cache Coherency Management (Cache Flush): Because the VC8000 relies on DMA to directly read from external memory, the program must execute dcache_clean_by_mva before invoking the hardware decoder. This action forces the video data temporarily stored in the CPU Cache to be flushed to the physical DDR memory. It prevents the hardware from reading outdated, un-updated data, which would otherwise lead to visual artifacts (image corruption).

3. Dynamic Buffer Update: After the hardware completes its execution, it returns the number of bytes that remain unconsumed (remain). The program uses this value to precisely calculate the actual amount of data consumed by the hardware (consumed) and synchronously updates the read pointer of the Ring Buffer. This maintains a stable and balanced data flow throughout the system.

```
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 4KB 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. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - 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 Hardware Requirements

- Circuit components
  - NuMaker-HMI-MA35D1-S1 V1.0

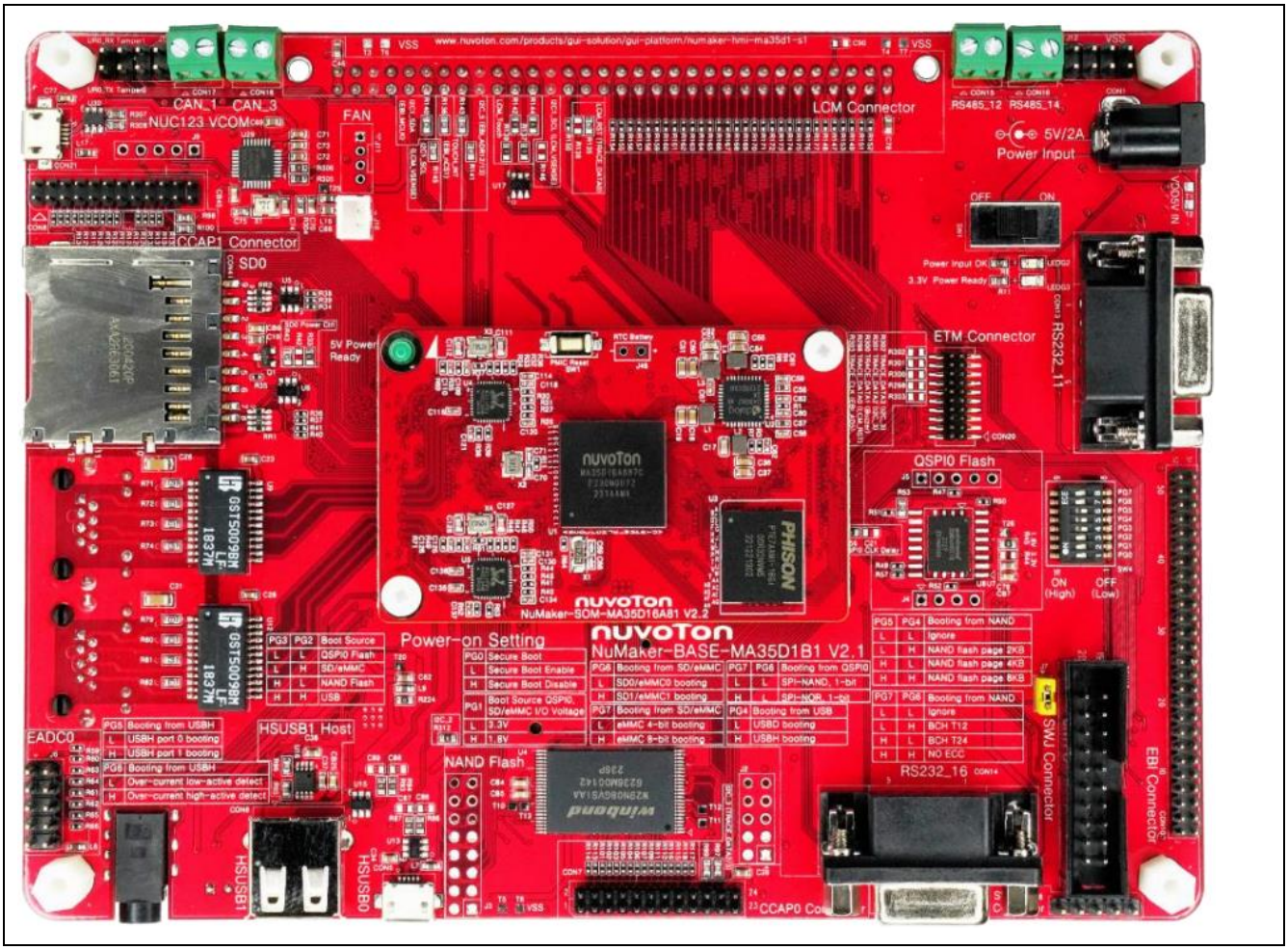


Figure 3-1 NuMaker-HMI-MA35D1-S1 V1.0

4. Directory Information

The directory structure is shown below.

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

Figure 4-1 Directory Structure

5. Example Code Execution

Once the sample program is successfully flashed as described in section 1.3, the system will begin listening for and awaiting H.264 RTP stream packets. To send video packets to the development board's IP address, you can either use a physical IP Camera or simulate the transmission using command-line tools on a PC.

The following instructions detail how to use the FFmpeg (version 4.2.7-0ubuntu0.1) on a PC (using an Ubuntu system as an example) to simulate an IPCAM sending RTP packets:

1. Prepare the Test Video: Please prepare a test video in H.264 format (this example uses 1024600.mp4). Place the video file in the current working directory on the Ubuntu system where you intend to execute the command.

2. Execute the FFmpeg Streaming Command: Open the Ubuntu Terminal and enter the following command. This command reads the video at its native frame rate and transmits it to the development board via the RTP protocol.

```
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. Verify the Streaming Status: After executing the command, the Ubuntu terminal will continuously output the current streaming progress and transmission rate. Please refer to Figure 5-1 for an illustration of the PC screen. At this point, the development board should be successfully receiving the RTP packets and initiating hardware decoding and video display.

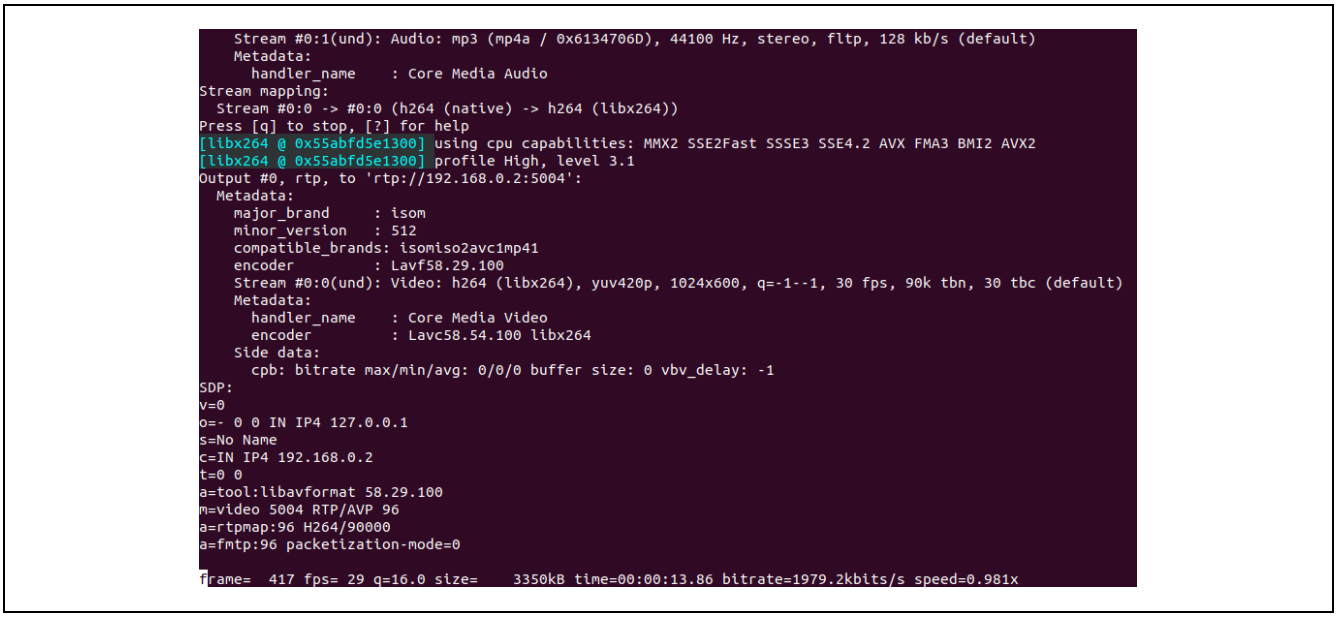


Figure 5-1 Test Command Screen

6. Revision History

Date	Revision	Description
2026.05.11	1.00	Initial version.

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.