M55M1 USB UVC

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code demonstrates the implementation of USB UVC (USB Video Device Class) functionality on the M55M1 development platform.
BSP Version	M55M1_Series_BSP_CMSIS_V3.00.001
Hardware	NuMaker-M55M1 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 example code demonstrates the integration of a UVC device (UVC, USB Video Class) with the NuMaker-M55M1 V1.0 evaluation board, connecting the HM1055 CMOS image sensor to capture images. The UVC device functions as a plug-and-play camera, requiring no driver installation when connected to a PC. (This Example Code uses the UVC 1.0 version.)

1.1 M55M1 Development Platform

The NuMicro® M55M1 series microcontroller excels in digital signal processing and on-device machine learning inference with its outstanding performance. It is equipped with an Arm® Cortex®-M55 processor running at up to 220 MHz. In addition to the Cortex-M55 core, it integrates the Arm® Ethos®-U55 Neural Processing Unit (NPU) as a coprocessor to accelerate neural network operations. The NuMaker-M55M1 V1.0 evaluation board offers a wide range of communication and peripheral interfaces, including the Universal Serial Control Interface (USCI), which can be flexibly configured as UART/SPI/I^{2}C, USB 2.0, Ethernet MAC with RMII, and a CCAP interface, as shown in Figure 1-1.

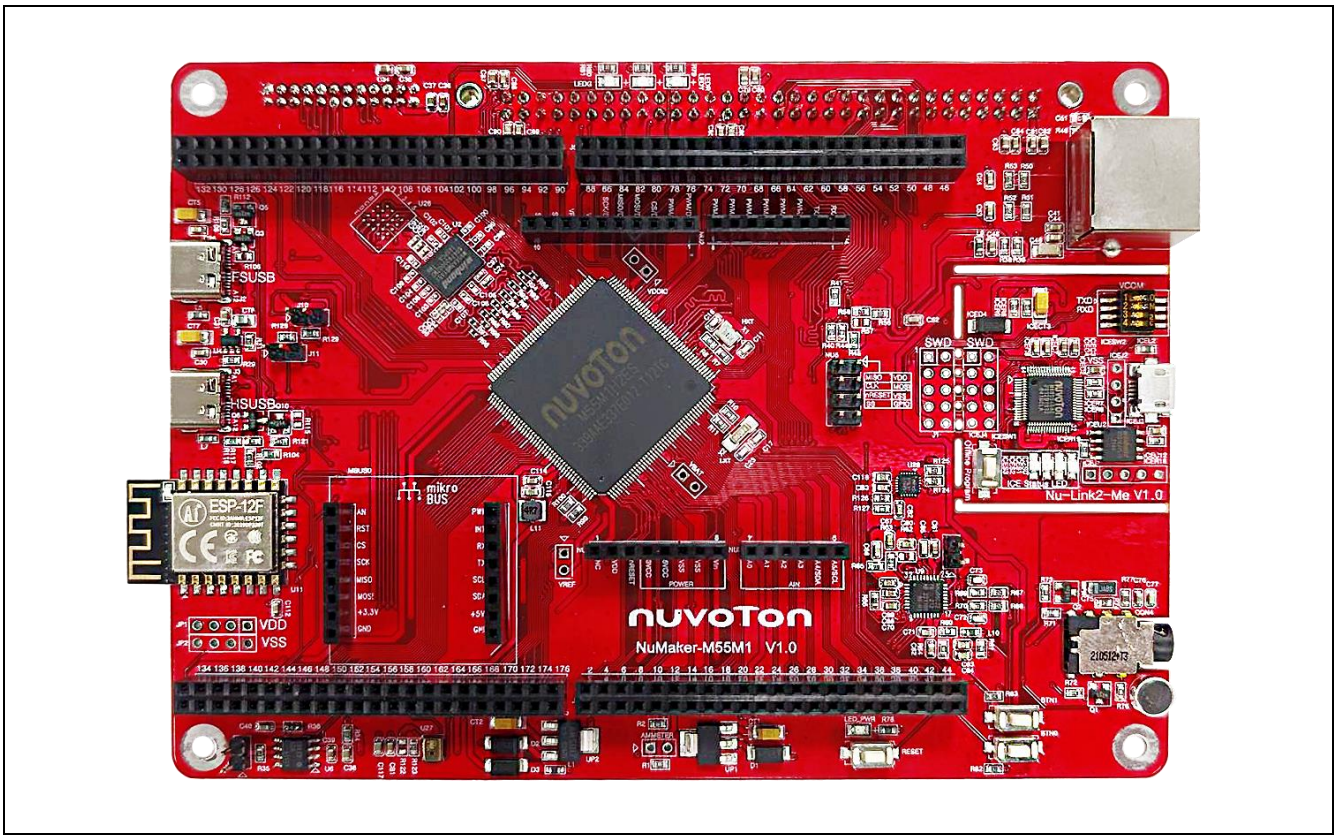

Figure 1-1 NuMaker-M55M1 V1.0 Evaluation Board

1.2 Image Sensor

This example uses the HM1055 CMOS image sensor, which provides high-quality images with excellent sensitivity and low noise, making it ideal for various imaging applications. It features a one-megapixel area scan sensor with an image size of 1280x720. As shown in Figure 1-2,

the HM1055 CMOS image sensor can be easily connected by installing the sensor onto the CCAP connector of the M55M1 development platform to complete the hardware setup.

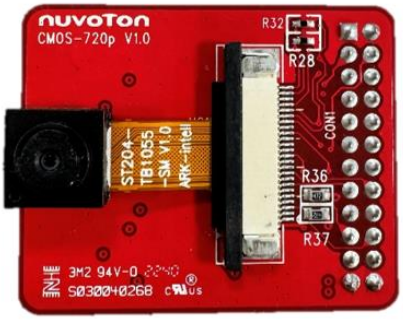

Figure 1-2 HM1055 CMOS Image Sensor

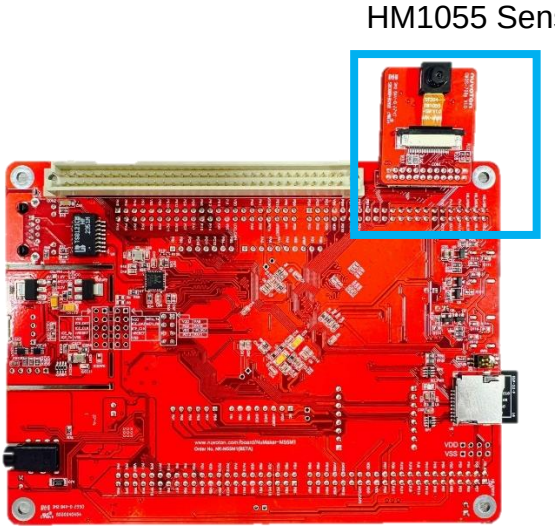

Figure 1-3 NuMaker-M55M1 V1.0 Connection with HM1055 CMOS Image Sensor Diagram

1.3 Analysis in Working Principles of UVC and USB Descriptors

1.3.1 Background Introduction

The image processing and transmission system begins by capturing images from the CMOS sensor, processes them through a processing module, and finally transmits them via USB to a PC for display. This process involves several key components and functions, including the CMOS sensor (such as the HM1055), the image processing module (CCAP), and the USB transmission module. USB Video Class (UVC) is a standard class specification that standardizes video streaming functions over USB, enabling devices such as webcams, digital cameras, analog video converters, and both analog and digital TV tuners to seamlessly connect with a host system. UVC supports streaming in various video formats, including YUV, MJPEG, MPEG-2 TS, H.264, and DV. It provides a structure for describing the capabilities of video devices and defines the methods for USB to control different device parameters and video stream characteristics based on requests.

UVC also offers flexibility in supporting multiple video resolutions, formats, and frame rates, which is crucial for bandwidth negotiation between the device and the host. Most operating system platforms support UVC drivers, significantly reducing the development time required for creating USB video devices.

1.3.2 UVC Module Diagram

Figure 1-4 Architecture Diagram

The camera system consists of a camera sensor module that captures real-world images using optical instruments, a UVC device that processes this data and adheres to UVC specifications, and a PC that identifies the firmware, loads the appropriate driver, communicates with the firmware, and enables the transmission and display of the captured camera data.

- Camera sensor module: uses optical instruments to capture real-world images.
- UVC Device: Processes the captured data and implements UVC-related specifications.
- PC: Recognizes the firmware information, loads the driver, communicates with the firmware, and facilitates the transmission and display of camera data.

1.3.3 Image Processing and Transmission Process

The entire image processing and transmission process begins with the initialization of the CMOS sensor (Camera Sensor Module). In the Sensor_HM1055.c file, the initialization of the CMOS sensor is handled by the InitSensor() function. This function is responsible for setting the sensor's output format, ensuring that the sensor can properly capture image data. Once initialized, the sensor begins capturing image data, which is digitized through the CaptureFrame() function, preparing it for the next stage of processing.

The captured image data then enters the M55M1 CCAP module for processing. During this stage, the PacketFormatDownScale() function adjusts the image data to the required format and size. The CCAP_Start() function is used to initiate the image capture interface, while the CCAP_SetPacketBuf() and CCAP_SetPacketScaling() functions store the data in memory, preparing it for subsequent USB transmission.

Once the image data is ready, the next step is to package and transmit it via USB. In the *uvc.c* file, this process is handled by the UVC_SendImage() function. First, this function resets the DMA transfer and then sets the appropriate data header depending on whether it's a static image or a video preview. The data is then sent to the host using the UVC_IsoIn() function via isochronous transfer. The UVC_ActiveDMA() function is used to initiate the USB DMA transfer, ensuring stable data transmission to the host.

Once the data reaches the host, the operating system and drivers are responsible for reassembling the data packets and displaying them as a continuous video stream. This process is primarily handled on the host side through the operating system's drivers and software.

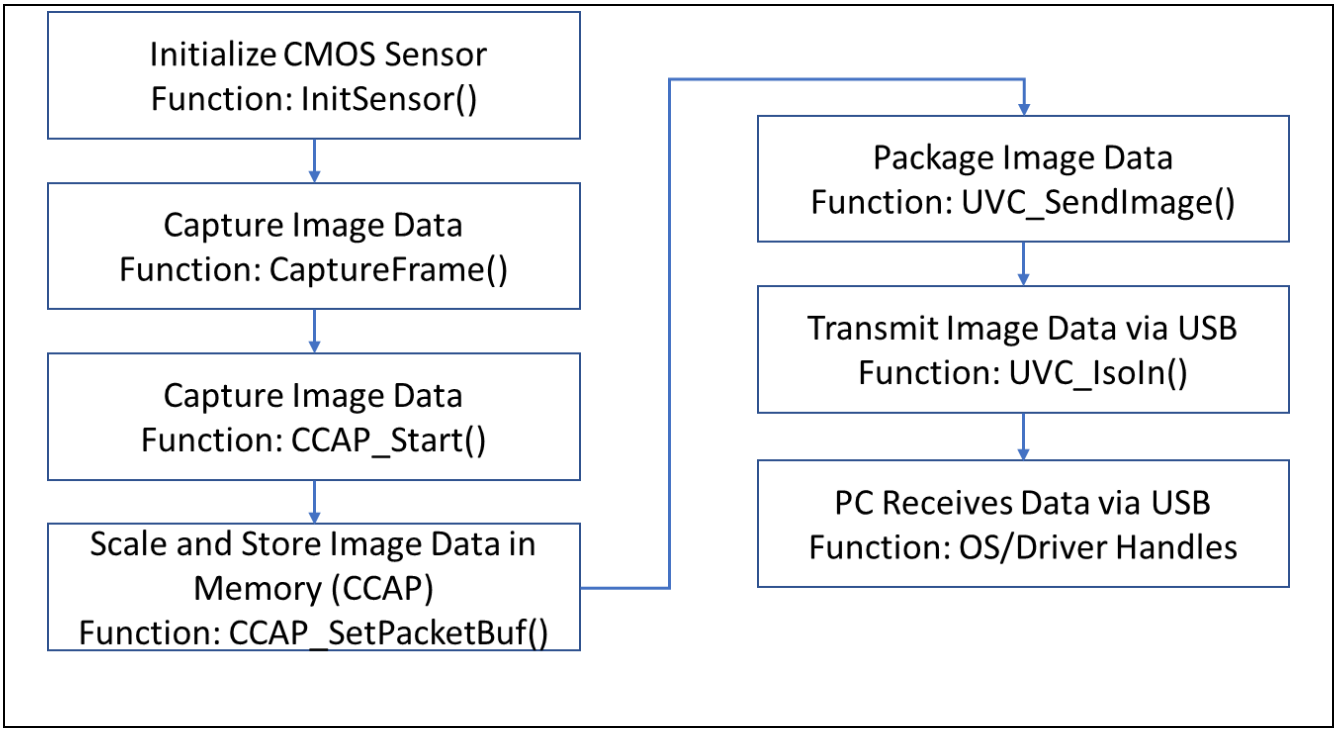

Figure 1-5 Program Flowchart

1.3.4 UVC Concept

UVC (USB Video Class) is a standard USB device type used for handling video data streams, such as transmitting video from a webcam to a computer. UVC devices interact with the host using standard protocols, and many operating systems natively support UVC, eliminating the need for specific drivers.

1.3.5 USB Descriptors Overview

USB descriptors are a set of structured data used to describe the attributes, functionalities, and requirements of a USB device. These descriptors help the host identify the device type, available features, and transmission needs. The main descriptors include:

- Device Descriptor: Describes the basic information of the USB device, such as device type, manufacturer, and product ID.
- Configuration Descriptor: Describes the device's configuration, including power requirements, number of interfaces, etc.
- Interface Descriptor: Describes the various interfaces of the device, such as video control interfaces or video streaming interfaces.
- Endpoint Descriptor: Defines the specific endpoints and methods for data transmission.

1.3.6 Role of Descriptors in UVC Devices

In UVC devices, descriptors provide detailed information on how the device interacts with the host and transmits video data. The following outlines the specific role of each descriptor in video transmission and its implementation in the code.

1.3.6.1 Device Descriptor

The device descriptor provides basic information about the USB device, such as device type, vendor ID, product ID, and the supported USB version. This is the first step for the host to obtain device information, allowing the host to identify the basic attributes of the device.

```
const uint8_t gu8DeviceDescriptor[] = {
    0x12,                   // bLength: The length of the descriptor. ( 18 bytes )
    USB_DT_DEVICE,          // bDescriptorType: Descriptor type ( Device )
    0x10, 0x01,             // bcdUSB: USB version ( USB 1.1 )
    0x00,                   // bDeviceClass: Device class ( Defined by interface )
    0x00,                   // bDeviceSubClass: Subclass
    0x00,                   // bDeviceProtocol: Protocol
    EP0_MAX_PKT_SIZE,       // bMaxPacketSize0: Maximum packet size for endpoint 0
    LOBYTE(USBD_VID),       // idVendor: Vendor ID of the device
    HIBYTE(USBD_VID),       // idProduct: Product ID of the device
    0x00, 0x01,             // bcdDevice: Device version number
    0x01,                   // iManufacturer: Manufacturer string index
    0x02,                   // iProduct: Product string index
    0x03,                   // iSerialNumber: Serial number string index
    0x01                    // bNumConfigurations: Number of supported configurations
};
```

1.3.6.2 Configuration Descriptor

The configuration descriptor describes the device's configuration options, including power usage and the number of supported interfaces. The host uses this descriptor to configure the device so that it can function properly.

Code Implementation:

```
const uint8_t gu8ConfigDescriptor[] = {
    0x09,                   // bLength: Length of the descriptor (9 bytes)
    USB_DT_CONFIG,          // bDescriptorType: Descriptor type (Configuration)
    USB_CONFIG_DESC_SIZE,   // wTotalLength: Total length, including all sub-descriptors
    0x02,                   // bNumInterfaces: Number of supported interfaces
    0x01,                   // bConfigurationValue: Configuration value (used to select
this configuration)
    0x00,                   // iConfiguration: Index of the configuration description
string
    0x80,                   // bmAttributes: Configuration characteristics (e.g., bus-
powered)
    0x32                    // MaxPower: Maximum current (100mA)
};
```

bNumInterfaces indicates that the configuration supports two interfaces: Video Control Interface and Video Streaming Interface.

1.3.6.3 Interface Descriptor

The interface descriptor provides detailed information about each interface in the configuration, including the functions provided (such as video control or video streaming) and the number of endpoints used.

```
const uint8_t gu8InterfaceDescriptor[] = {
    0x09,                   // bLength: Length of the descriptor (9 bytes)
```

```
    USB_DT_INTERFACE,       // bDescriptorType: Descriptor type (Interface)
    0x00,                   // bInterfaceNumber: Interface number
    0x00,                   // bAlternateSetting: Alternate setting
    0x01,                   // bNumEndpoints: Number of endpoints used by this interface
(1 endpoint)
    USB_CLASS_VIDEO,        // bInterfaceClass: Interface class (Video class)
    USB_SUBCLASS_VIDEOCONTROL, // bInterfaceSubClass: Subclass (Video control)
    USB_PROTOCOL_UNDEFINED, // bInterfaceProtocol: Protocol
    0x00                    // iInterface: Index of the interface description string
};
```

Here, 'bInterfaceClass' is set to 'USB_CLASS_VIDEO', indicating that it is a video class interface.

'bNumEndpoints' is set to 1, signifying that the interface uses only one endpoint for data transmission.

1.3.6.4 Endpoint Descriptor

The endpoint descriptor defines the specific method of data transmission between the host and the device. For video streaming, endpoints are typically configured for isochronous transfer to ensure data stability and real-time transmission.

```
const uint8_t gu8EndpointDescriptor[] = {
    0x07,                   // bLength: Length of the descriptor (7 bytes)
    USB_DT_ENDPOINT,        // bDescriptorType: Descriptor type (Endpoint)
    ISO_IN_EP_NUM,          // bEndpointAddress: Endpoint address (IN endpoint)
    USB_ENDPOINT_TYPE_ISOCHRONOUS, // bmAttributes: Endpoint attributes (Isochronous
transfer)
    LOBYTE(MAX_ISO_PACKET_SIZE),  // wMaxPacketSize: Maximum packet size
    HIBYTE(MAX_ISO_PACKET_SIZE),
    0x01                    // bInterval: Polling interval ( 1ms )
};
```

bEndpointAddress indicates that it is an IN endpoint, used for transferring data from the device to the host.

bmAttributes is set to USB_ENDPOINT_TYPE_ISOCHRONOUS, indicating that it is an isochronous endpoint suitable for video streaming.

1.4 Complete Process of CMOS Image Data Transmission

1.4.1 CMOS Image Capture

The CMOS sensor is responsible for capturing the image and digitizing it, which is the first step in the entire video transmission process.

In Sensor_HM1055.c, the part initializes and starts image capture:

```
/* Initialize sensor and set output format as YUV422 */
if (psSensorInfo->pfnInitSensor(0) == FALSE) {
    printf("Init sensor failed !\n");
    return -1;
}
```

1.4.2 Image Processing and Data Packaging

The captured image data needs to be processed and packaged into packets that can be transmitted via USB. This includes data scaling and format conversion, and finally storing the data in memory, ready for transmission.

In the PacketFormatDownScale() function:

```
/* Set System Memory Packet Base Address Register */
CCAP_SetPacketBuf((uint32_t)u8FrameBuffer0);

/* Set Packet Scaling Vertical/Horizontal Factor Register */
CCAP_SetPacketScaling(SYSTEM_HEIGHT, psSensorInfo->m_u16Height, SYSTEM_WIDTH,
psSensorInfo->m_u16Width);

/* Set Packet Frame Output Pixel Stride Width */
CCAP_SetPacketStride(SYSTEM_WIDTH);

/* Start Image Capture Interface */
CCAP_Start();
```

This code scales the captured image data and prepares the memory address, making it ready for subsequent USB transmission.

1.4.3 Data Transfer to Host

The image data is packaged and transmitted to the host via USB. Isochronous transfer mode is used here to ensure the real-time performance of the video stream.

In the UVC_SendImage() function:

```
HSUSBD_ResetDMA();

if (bStillImage) {
    g_u8UVC_PD = UVC_PH_Snapshot;  /* Payload Header for Snapshot */
    uvcStatus.StillImage = 0;
} else {
    g_u8UVC_PD = UVC_PH_Preview;  /* Payload Header for Preview */
}

UVC_IsoIn(u32Addr, u32transferSize);
```

The ‘UVC_SendImage()’ function is responsible for transmitting data to the host, with the ‘UVC_IsoIn()’ function sending the data via USB isochronous transfer. The input parameter ‘u32Addr’ represents the address of the image data, while ‘u32transferSize’ specifies the size of the data to be transmitted.

1.4.4 Data Transmission to Host

After the host receives these data packets, it reassembles the data according to the UVC protocol and displays the video. At the code level, this process is primarily handled by the host's operating system and drivers, so the specific code is not visible on the device side.

In the UVC_IsoIn() function:

```
HSUSBD_SET_DMA_READ(ISO_IN_EP_NUM);
UVC_ActiveDMA(u32Addr, u32Len);
```

The UVC_ActiveDMA() function activates the USB DMA transfer, which is a crucial step in transmitting data to the host.

1.5 Detailed Explanation of Image Data Packaging Process

Based on the USB UVC packet content, it is divided into multiple fields, each requiring corresponding data to be filled in and then packed for transmission to the PC. The explanation of each field in the packet is as follows:

- UVC Payload Header: This is the header of the packet, containing control information for the frame.
- bHeaderLength: Describes the length of the UVC Payload Header.
- bmHeaderInfo: Contains frame flag bits, indicating whether this is the start or end of a frame, and can also mark the type of frame (e.g., still image).
- dwPresentationTime: A 4-byte timestamp used to synchronize the playback time of the image.
- dwSourceClock: A 4-byte clock timestamp used to synchronize the capture time of the image.
- Image Data Payload: This is the main content of the packet, containing the actual image data.
- Video Data: This part stores the captured image data, with its size depending on the image resolution and frame format.
- USB Packet Footer: This part is optional and used to store additional information, such as error check codes (like CRC).
- Error Check (CRC): If supported by the system, an error check code can be added at the end of the packet to ensure data integrity.

1.6 Corresponding Packet Structure in the Code

In the code, these components are handled by the ‘UVC_SendImage()’ function for packaging and transmission. For example, ‘UVC_PH_Preview’ and ‘UVC_PH_Snapshot’ correspond to ‘bmHeaderInfo’, while the image data is read from the image buffer and placed into the packet.

```
if (bStillImage) {
    g_u8UVC_PD = UVC_PH_Snapshot;  /* Payload Header for Snapshot */
    uvcStatus.StillImage = 0;
} else {
    g_u8UVC_PD = UVC_PH_Preview;  /* Payload Header for Preview */
}
```

This code sets the ‘bmHeaderInfo’ in the UVC Payload Header and determines the packet type based on whether it is a still image or video.

1.7 UVC Frame Rate

The UVC program takes 75 ms to transmit one image to the PC, with an approximate frame rate of 13 FPS (frame per second).

```
while (1)
{

    u32TimeVal = g_u32Ticks;

    if (gAppConnected == 1)
    {
        if (s_u32XferBufAddr) //Trigger ISO-IN DMA
        {
            UVC_SendImage(s_u32XferBufAddr, s_u32XferSize, uvcStatus.StillImage);
            s_u32XferBufAddr = 0;  // Set ISO-IN IDLE
        }
    }

    u32TimeVal_Vec = g_u32Ticks;

printf("Time = %d ms\n", (u32TimeVal_Vec - u32TimeVal) / 1  );
```

Results of the UVC frame rate:

```
Time = 75 ms
```

2. Code Description

The main program is as follows: after initializing the system, it sets up the HM1055 and starts image capturing, followed by setting up the USB Device.

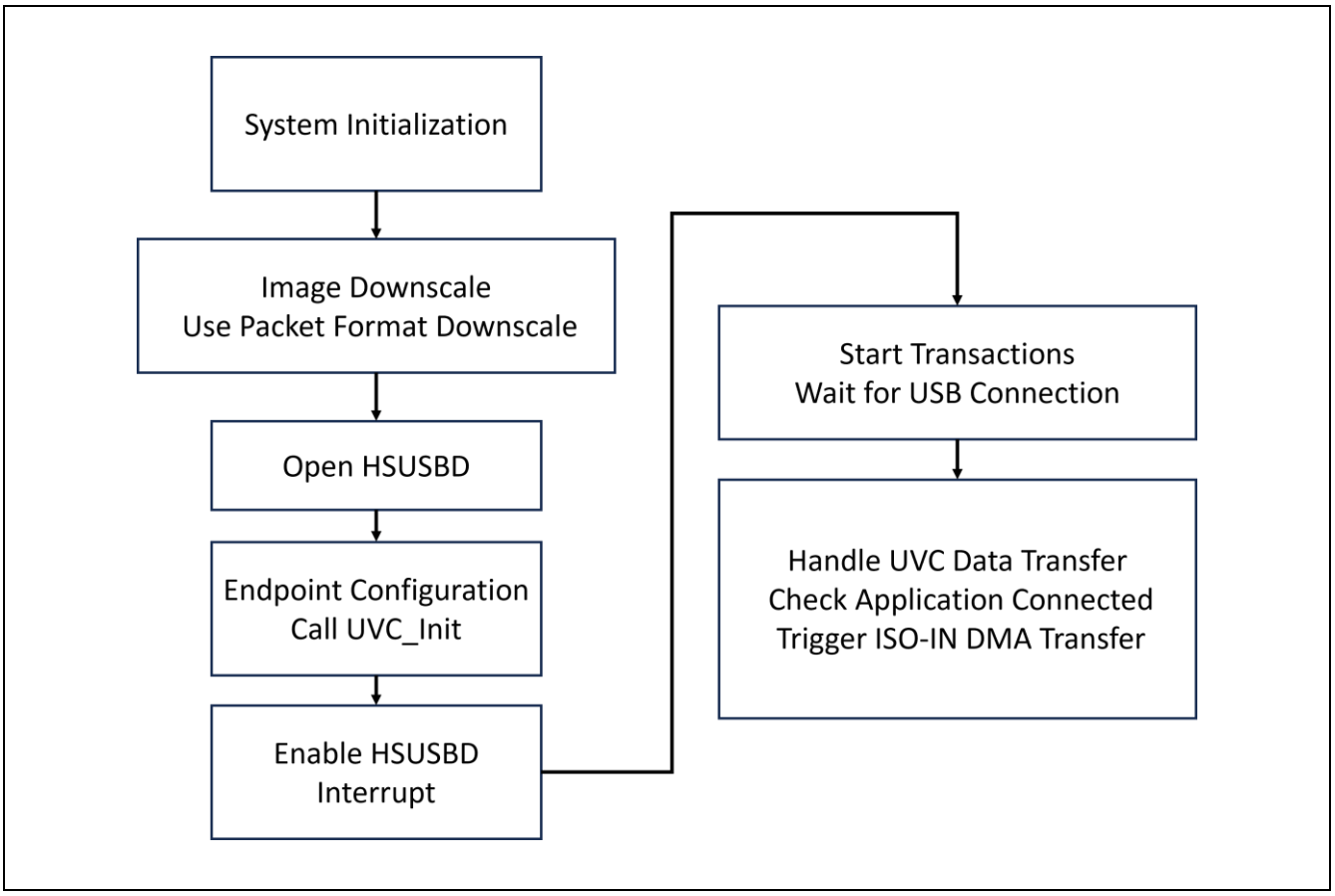

Figure 2-1 Program Flowchart

```
int32_t main(void)
{
    /* Init System, peripheral clock and multi-function I/O */
    SYS_Init();

    /* Init UART to 115200-8n1 for print message */
    InitDebugUart();

    printf("NuMicro HSUSBD UVC YUV422 Streaming\n");

    /* Using Packet format to Image down scale */
    if (PacketFormatDownScale(&g_sSensorHM1055) != 0)
        printf("Capture frame failed !\n");

    HSUSBD_Open(&gsHSInfo, UVC_ClassRequest, UVC_SetInterface);

    /* Endpoint configuration */
    UVC_Init();
```

```
    /* Enable HSUSBD interrupt */
    NVIC_EnableIRQ(HSUSBD_IRQn);

    /* Start transaction */
    while (1)
    {
        if (!HSUSBD_IS_ATTACHED())
        {
            HSUSBD_Start();
            break;
        }
    }

    while (1)
    {
        if (gAppConnected == 1)
        {
            if (s_u32XferBufAddr) //Trigger ISO-IN DMA
            {
                UVC_SendImage(s_u32XferBufAddr, s_u32XferSize, uvcStatus.StillImage);
                s_u32XferBufAddr = 0; // Set ISO-IN IDLE
            }
        }
    }
}
```

The setup and initialization of the HM1055 configure image capture (CCAP) with the output format set to YUV422:

```
int32_t PacketFormatDownScale(S_SENSOR_INFO *psSensorInfo)
{
    uint32_t u32Frame;

    /* Init CCAP clock and Sensor clock */
    CCAP_SetFreq(12000000, 12000000);

    /* Initialize sensor and set output format as YUV422 */
    if (psSensorInfo->pfnInitSensor(0) == FALSE)
    {
        printf("Init sensor failed !\n");
        return -1;
    }

    /* Enable External CCAP Interrupt */
    NVIC_EnableIRQ(CCAP_IRQn);

    /* Enable CCAP Frame End Interrupt */
    CCAP_EnableInt(CCAP_INT_VIEN_ENABLE);

    /* Set Vsync polarity, Hsync polarity, pixel clock polarity, Sensor Format and Order
*/
    CCAP_Open(psSensorInfo->m_u32Polarity, psSensorInfo->m_u32InputFormat,
CCAP_PAR_OUTFMT_YUV422);
```

```
    /* Set Cropping Window Vertical/Horizontal Starting Address and Cropping Window Size
*/
    CCAP_SetCroppingWindow(0, 0, psSensorInfo->m_u16Height, psSensorInfo->m_u16Width);

    /* Set System Memory Packet Base Address Register */
    CCAP_SetPacketBuf((uint32_t)u8FrameBuffer0);

    /* Set Packet Scaling Vertical/Horizontal Factor Register */
    CCAP_SetPacketScaling(SYSTEM_HEIGHT, psSensorInfo->m_u16Height, SYSTEM_WIDTH,
psSensorInfo->m_u16Width);

    /* Set Packet Frame Output Pixel Stride Width */
    CCAP_SetPacketStride(SYSTEM_WIDTH);

    /* Start Image Capture Interface */
    CCAP_Start();

    return 0;
}
```

The ChangeFrame function is used to modify the video input settings, adjusting the frame size.

```
/*---------------------------------------------------------------------------------------*/
/* Change VideoIN Setting for Frame size */
/*---------------------------------------------------------------------------------------*/

void ChangeFrame(char bChangeSize, uint32_t u32Address, uint16_t u16Width, uint16_t
u16Height)
{
    if (bChangeSize)    /* Change Frame Size */
    {
        CCAP_SetPacketScaling(u16Height, SENSOR_IN_HEIGHT, u16Width, SENSOR_IN_WIDTH);
    }
    /* Set Buffer Address */
    CCAP_SetPacketBuf((uint32_t)u32Address);
    CCAP_SetPacketStride(u16Width);
}
```

The ChangeFrame function is used to modify the video input settings, adjusting the frame size.

```
void UVC_Init(void)
{
    /* Configure USB controller */
    /* Enable USB BUS, CEP and EPA global interrupt */
    HSUSBD_ENABLE_USB_INT(HSUSBD_GINTEN_USBIEN_Msk | HSUSBD_GINTEN_CEPIEN_Msk |
HSUSBD_GINTEN_EPAIEN_Msk | HSUSBD_GINTEN_EPCIEN_Msk);
    /* Enable BUS interrupt */
    HSUSBD_ENABLE_BUS_INT(HSUSBD_BUSINTEN_DMADONEIEN_Msk | HSUSBD_BUSINTEN_RESUMEIEN_Msk |
HSUSBD_BUSINTEN_RSTIEN_Msk | HSUSBD_BUSINTEN_VBUSDETIEN_Msk);
    /* Reset Address to 0 */
    HSUSBD_SET_ADDR(0);


    /* Control endpoint */
```

```
    HSUSBD_SetEpBufAddr(CEP, CEP_BUF_BASE, CEP_BUF_LEN);
    HSUSBD_ENABLE_CEP_INT(HSUSBD_CEPINTEN_SETUPPKIEN_Msk |
HSUSBD_CEPINTEN_STSDONEIEN_Msk);

    UVC_InitForHighSpeed();

    uvcInfo.u8ErrCode = 0;
    uvcInfo.bChangeData = 0;
    uvcStatus.StillImage = 0;

    uvcInfo.IsoMaxPacketSize[0] = gMaxPacketSize;
    uvcInfo.IsoMaxPacketSize[1] = gMaxPacketSize;
    uvcInfo.IsoMaxPacketSize[2] = gMaxPacketSize;

    //Initialize Video Capture Filter Control Value
    uvcInfo.CapFilter.Brightness = 1;
    uvcInfo.CapFilter.POWER_LINE_FREQUENCY = 2;
    uvcInfo.CapFilter.Brightness = 1;
    uvcInfo.CapFilter.Contrast = 2;
    uvcInfo.CapFilter.Hue = 3;
    uvcInfo.CapFilter.Saturation = 4;
    uvcInfo.CapFilter.Sharpness = 5;
    uvcInfo.CapFilter.Gamma = 3;
    uvcInfo.CapFilter.Backlight = 7;

    /* Initialize Video Probe and Commit Control data */
    memset((uint8_t *)&uvcInfo.VSCmdCtlData, 0x0, sizeof(VIDEOSTREAMCMDDATA));
    uvcInfo.VSCmdCtlData.bmHint = 0x0100;
    uvcInfo.VSCmdCtlData.bFormatIndex = 1;
    uvcInfo.VSCmdCtlData.bFrameIndex = 2;
    uvcStatus.FormatIndex = 1;
    uvcStatus.FrameIndex = 2;
    uvcInfo.VSCmdCtlData.dwFrameInterval = 0x051615;

    /* Initialize Still Image Command data */
    memset((uint8_t *)&uvcInfo.VSStillCmdCtlData, 0x0, sizeof(VIDEOSTREAMSTILLCMDDATA));

    uvcInfo.VSStillCmdCtlData.bFormatIndex = 1;
    uvcInfo.VSStillCmdCtlData.bFrameIndex = 1;
    uvcStatus.snapshotFormatIndex = 1;
    uvcStatus.snapshotFrameIndex = 1;
}
```

After capturing the image, it is transmitted to the PC via USB.

```
int UVC_SendImage(uint32_t u32Addr, uint32_t u32transferSize, uint32_t bStillImage)
{
    if (u32transferSize != 0)
    {

        uvcStatus.bReady = 0;
        HSUSBD_ResetDMA();

        if (bStillImage)
        {
            g_u8UVC_PD = UVC_PH_Snapshot;       /* Payload Header for Snapshot */
```

```
            uvcStatus.StillImage = 0;
        }
        else
            g_u8UVC_PD = UVC_PH_Preview;        /* Payload Header for Preview */

        UVC_IsoIn(u32Addr, u32transferSize);

        return 1;
    }
    return 0;
}
```

The ‘CCAP_IRQHandler’ is an interrupt service routine used to handle interrupts from the image capture module (CCAP).

In the following code, you can see the switching between two frame. The two frame buffers (u8FrameBuffer0 and u8FrameBuffer1) are used to implement a double-buffering mechanism. The main purpose of this is to prevent image tearing and enhance the stability of image processing. While one buffer is used to display or transfer the current image, the other buffer is responsible for receiving and writing the new frame data. This ensures that the same buffer is not read from and written to simultaneously, preventing the screen from being modified during reading, which would result in tearing. By alternating between the two buffers, the system achieves smooth frame transitions, avoiding the display of partial new and partial old data. This also improves data transfer efficiency, ensuring the continuity and fluidity of image processing.

When u32UVCXferIdx is odd, the data is written to u8FrameBuffer1, and the next frame is set to use u8FrameBuffer0.

When u32UVCXferIdx is even, the data is written to u8FrameBuffer0, and the next frame is set to use u8FrameBuffer1.

```
/*---------------------------------------------------------------------------*/
/* To run CCAPInterruptHandler, when CAP frame end interrupt                 */
/*---------------------------------------------------------------------------*/
void CCAPInterruptHandler(void)
{
    /* UVC transfer counter */
    static uint32_t u32UVCXferIdx = 0;

    /* Current Frame Index */
    static uint32_t u32CurFrameIndex = 2;

    /* Current Format Index */
    static uint32_t u32CurFormatIndex = 1;

    /* Current Width & Height */
    static uint16_t u16CurWidth = 320, u16CurHeight = 240;

    uint32_t u32NextFrameBufferAddr;

    if (s_u32XferBufAddr == 0) // UVC ISO-IN IDLE
    {
        if (u32UVCXferIdx & 0x1u)
        {
```

```
            s_u32XferBufAddr = (uint32_t)u8FrameBuffer1;
            u32NextFrameBufferAddr = (uint32_t)u8FrameBuffer0;
        }
        else
        {
            s_u32XferBufAddr = (uint32_t)u8FrameBuffer0;
            u32NextFrameBufferAddr = (uint32_t)u8FrameBuffer1;
        }

        /* UVC_Foramt_YUY2 */
        switch (uvcStatus.FormatIndex)
        {
        case UVC_Format_YUY2:       /* UVC_Foramt_YUY2 */
        {
            s_u32XferSize = uvcStatus.MaxVideoFrameSize;
            if (u32CurFrameIndex != uvcStatus.FrameIndex || u32CurFormatIndex !=
uvcStatus.FormatIndex)
            {
                /* Frame Size Changed */
                switch (uvcStatus.FrameIndex)
                {
                case UVC_VGA:
                    u16CurWidth = 640;
                    u16CurHeight = 480;
                    break;
                case UVC_QVGA:
                    u16CurWidth = 320;
                    u16CurHeight = 240;
                    break;
                case UVC_QQVGA:
                    u16CurWidth = 160;
                    u16CurHeight = 120;
                    break;
                }
                /* Set VideoIn Setting for Frame Size */
                ChangeFrame(TRUE, u32NextFrameBufferAddr, u16CurWidth, u16CurHeight);
                u32CurFrameIndex = uvcStatus.FrameIndex;
                u32CurFormatIndex = uvcStatus.FormatIndex;
            }
            else
                ChangeFrame(FALSE, u32NextFrameBufferAddr, u16CurWidth, u16CurHeight);

            break;
        }
        default:
            break;
        }

        u32UVCXferIdx++;
    }
}

/*---------------------------------------------------------------------------------------*/
/*  CCAP_IRQHandler                                                                      */
/*---------------------------------------------------------------------------------------*/
NVT_ITCM void CCAP_IRQHandler(void)
```

```
{
    uint32_t u32IntStatus = CCAP_GET_INT_STS();

    if (CCAP_IsIntEnabled(CCAP_INT_VIEN_ENABLE) && (u32IntStatus & CCAP_INTSTS_VINTF_Msk)
== CCAP_INTSTS_VINTF_Msk)
    {
        CCAPInterruptHandler();
        CCAP_CLR_INT_FLAG(CCAP_INTSTS_VINTF_Msk); /* Clear Frame end interrupt */
    }

    if (CCAP_IsIntEnabled(CCAP_INT_ADDRMIEN_ENABLE) && (u32IntStatus &
CCAP_INTSTS_ADDRMINTF_Msk) == CCAP_INTSTS_ADDRMINTF_Msk)
    {
        CCAP_CLR_INT_FLAG(CCAP_INTSTS_ADDRMINTF_Msk);  /* Clear Address match interrupt */
    }

    if (CCAP_IsIntEnabled(CCAP_INT_MEIEN_ENABLE) && (u32IntStatus &
CCAP_INTSTS_MEINTF_Msk) == CCAP_INTSTS_MEINTF_Msk)
    {
        CCAP_CLR_INT_FLAG(CCAP_INTSTS_MEINTF_Msk);    /* Clear Memory error interrupt */
    }

    CCAP_EnableUpdate();

    // CPU read interrupt flag register to wait write(clear) instruction completement.
    u32IntStatus = CCAP_GET_INT_STS();
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M55M1_Series_BSP_CMSIS_V3.00.001

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M55M1 V1.0
  - HM1055 CMOS Sensor

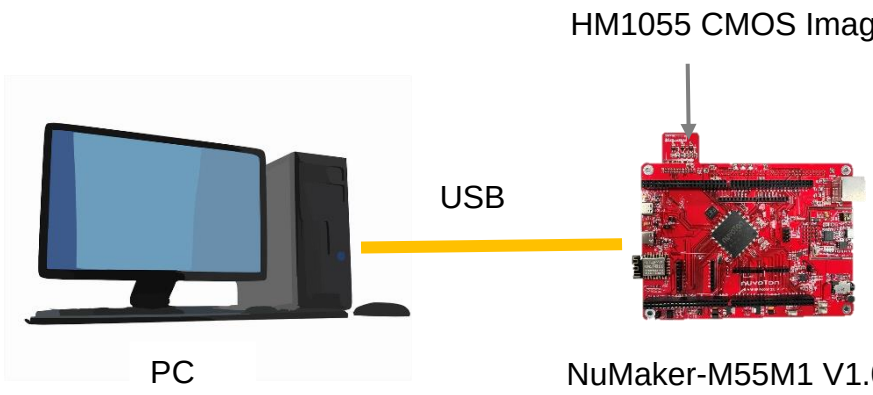

Figure 3-1 USB UVC Connection Execution Diagram

- CMOS Circuit Diagram
  - The M55M1 uses I^2C to transmit control commands to the CMOS sensor and uses CCAP_SCLK as the SCLK to supply frequency to the CMOS sensor. The CMOS sensor returns the Pixel Clock (PIXCLK), HSYNC, VSYNC, and D0~D7 data according to the settings.

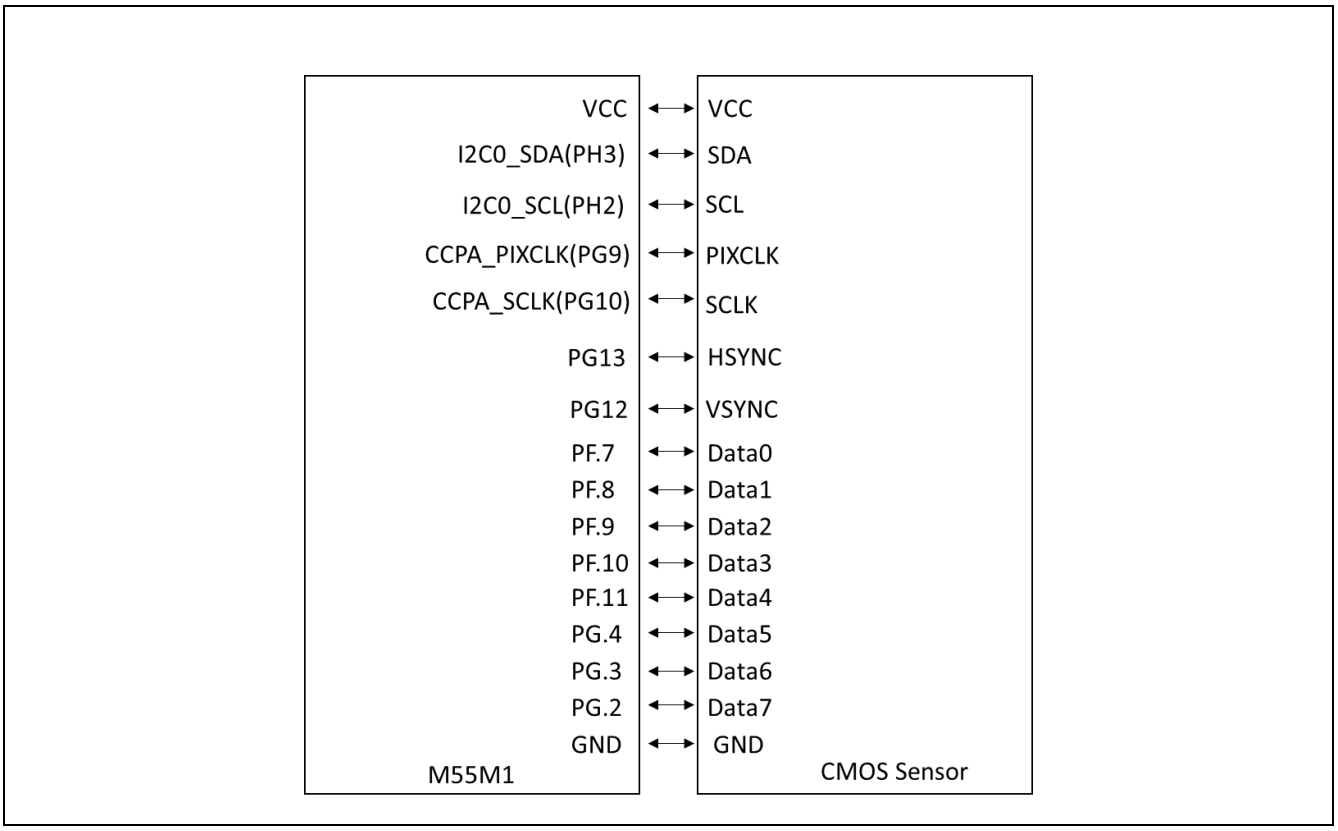

Figure 3-2 CMOS Circuit Diagram

4. Directory Information

The directory structure is shown below.

EC_M55M1_HSUSBD_UVC_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
SampleCode	
ExampleCode	
HSUSBD_UVC	Source file of example code

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *HSUSBD_UVC_VideoSensor.uvprojx*.
2. Enter the compilation mode interface:
   - Compile
   - Download code to memory
   - Enter/Exit debug mode
3. Enter the debug mode interface:
   - Run the code
   - Show the results

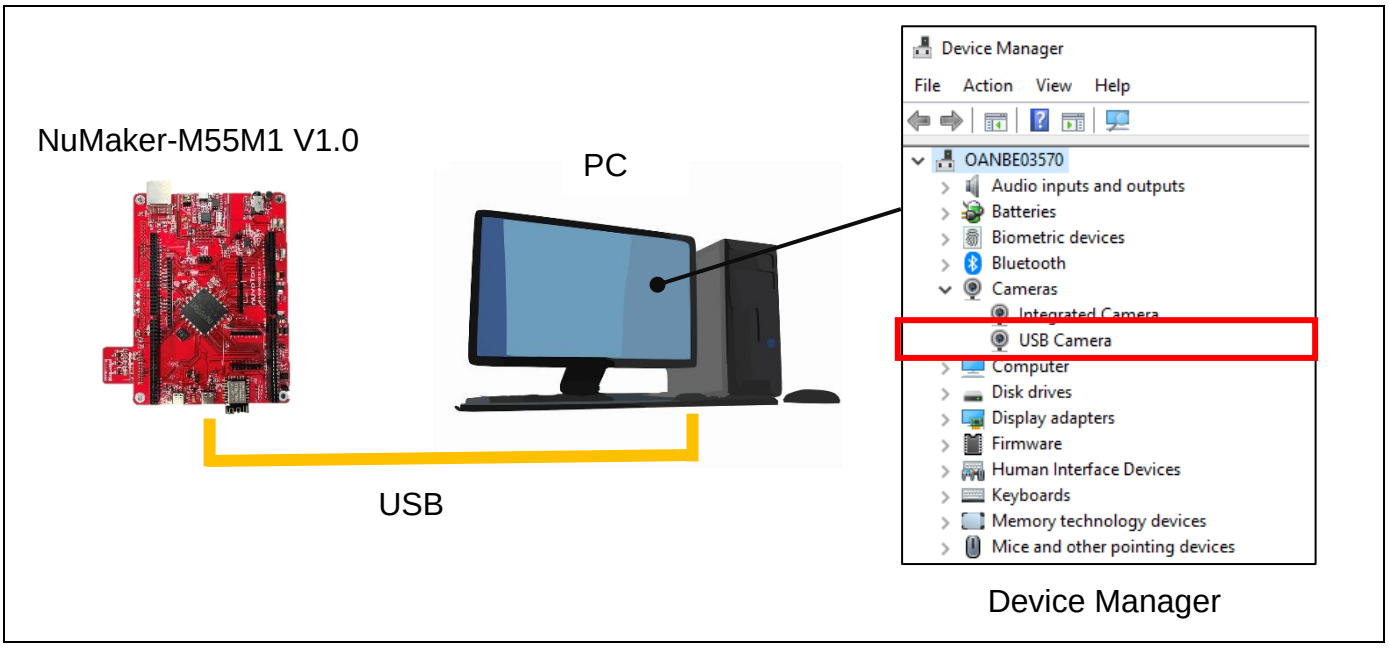

Figure 5-1 Results of the USB UVC

6. Revision History

Date	Revision	Description
2024.06.14	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.