

Capturing images from a USB camera using STM32

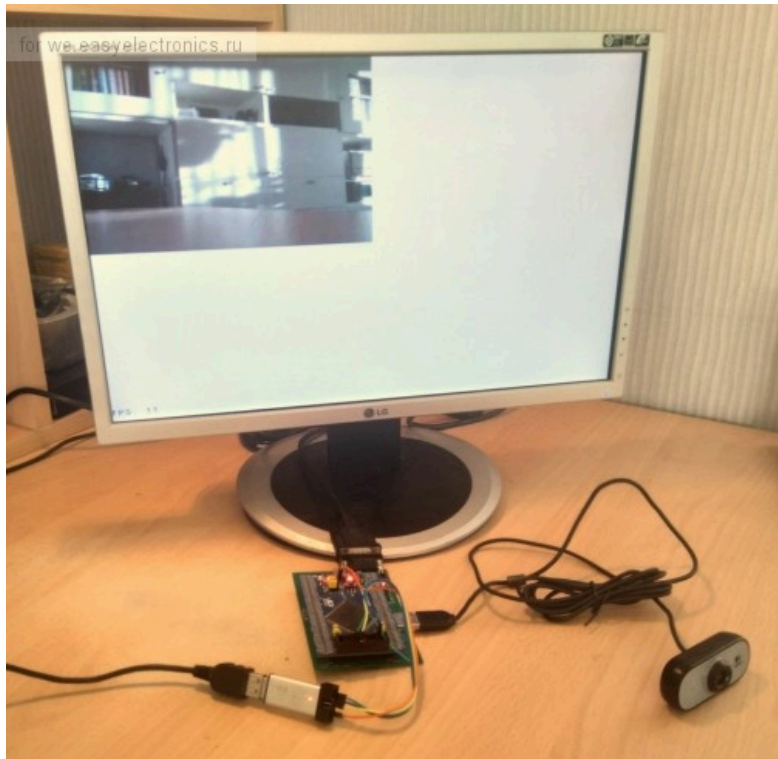
STM32

For my own self-study, I decided to connect a USB camera (webcam) to the STM32. I already had [a debug board based on the STM324F429](#), which could output images to a VGA monitor, so I used it to test the camera.

Which to choose: HAL or SPL?

It's clear that the USB controller needs to operate in Host mode. I hadn't worked with USB Host much before, and in this case, I absolutely had to use Isochronous Transfers mode, for which there are traditionally very few examples. However, the STM32 Cube can generate code for the USB Host Audio Class for the HAL, so I used that generated example as a basis. This example is designed for working with a USB audio card. To enable the USB driver code to generate debug messages, you need to set a constant:

```
#define USBH_DEBUG_LEVEL      3
```



After connecting the audio card, the microcontroller indeed detected it—the USB driver and audio class code output various debug messages to the Semihosting output window, indicating that the hardware was working properly. Next, I converted the audio class code to a UVC class.

It's worth noting that most USB video cameras operate using a special USB class called UVC (USB Video Class). [I've encountered it](#) before . I'll try to briefly describe what's happening in my program.

1. Analysis of descriptors

The first step is to obtain descriptors from the camera and analyze them.

The STM USB driver handles receiving descriptors from the camera, so the user only needs to analyze them. It's important that the "USBH_MAX_SIZE_CONFIGURATION" value is large enough (mine is 1024), otherwise the received descriptors simply won't fit in the controller's memory.

When analyzing descriptors, you need to check whether the connected device is UVC-compliant.

Also of interest in the descriptors are the various *interface* configuration options , specifically the interface number and endpoint size . The camera transmits several supported endpoint sizes, and we need to select the most appropriate one.

The allowed endpoint size is limited by the controller's hardware FIFO buffer. How this FIFO is allocated for different USB tasks is configured in the "stm32f4xx_ll_usb.c" file (search for "GRXFSIZ"). Unfortunately, the necessary constants are hardcoded into the code, so I had to modify this file to maximize the RX FIFO area.

Furthermore, the descriptors convey all possible camera image transmission modes. Among them, the "Format Type Descriptor" descriptors stand out—they describe the possible image transmission formats. We're interested in the most common ones: YUY2 (Uncompressed Format) and MJPEG (MJPEG Format). Each of these descriptors is followed by several "Frame Type" descriptors, each corresponding to a specific image size transmitted by the camera.

Each of these descriptors contains the "bFormatIndex" and "bFrameIndex" fields, whose values can be used to select the camera's operating mode.

Thus, for the user, the task of descriptor analysis boils down to finding the required descriptors in the data array received from the camera and determining the appropriate values for the endpoint address through which data from the camera will be transmitted, the interface number, bFormatIndex, and bFrameIndex.

Once all descriptors have been successfully analyzed, the required endpoint can be opened, which will receive data from the camera.

It's worth noting that the STM code example looks very confusing. It seems to support the OUT (data transmission from the host) and IN (data transmission to the host—the option we need), but in reality, the IN mode is only partially implemented. However, the developers crammed HID work (for volume control) into it, and crammed a huge pile of disparate code into a single file. I had to significantly rework this entire file; for example, I separated descriptor analysis and processing of received data into separate files.

The USB driver itself contains [a bug](#), disrupting the operation of Isochronous IN endpoints.

2. Setting up camera parameters

The next step is to pass certain parameters to the camera—specifically, "bFormatIndex" and "bFrameIndex"—to indicate the image size and mode it should transmit.

In the HAL, this process is performed in the USBH_UserProcess() function.

The protocol includes special requests: GET_CUR (for requesting current camera settings) and SET_CUR (for setting new camera parameters). Each of these requests must be of a specific type: PROBE_CONTROL (for trial parameter setting) and COMMIT_CONTROL (for confirming parameter setting).

Thus, the Host first sends certain settings to the camera (SET_CUR + PROBE_CONTROL), then reads them from the camera (GET_CUR + PROBE_CONTROL). At this stage, the camera must return adequate values, including those that were previously set. After this, the set of values must be confirmed by sending a request (SET_CUR + COMMIT_CONTROL).

3. Starting data transfer

At this stage, you need to send a ["Set Interface"](#) request to the camera. This is already implemented in the USB driver as the "USBH_SetInterface()" function. The "bInterfaceNumber" and "bAlternateSetting" values obtained earlier during descriptor analysis should be passed as parameters.

Upon receiving this request, the camera begins transmitting data to the host microcontroller.

4. Processing of received data

After the camera starts transmitting data, the controller's job is to process it.

User code must check at least every 1 ms for new data from the camera; if so, process it.

The HAL assumes that user data processing code should be called from the "BgndProcess" callback function, which, in turn, is called from the MX_USB_HOST_Process() function—the single HAL function for managing USB-Host. The HAL also assumes that the MX_USB_HOST_Process() function is located in the main loop (superloop) of the program. Clearly, any long-running function in the superloop blocks calls to MX_USB_HOST_Process(), and some USB data from the camera is lost. To avoid this, I had to move data processing from the camera to an interrupt from a preset timer.

The camera transmits data in isochronous packets. According to the UVC standard, each packet contains a header (usually 12 bytes long) and payload data. For us, only the second byte of the header is important—it contains bit flags that carry information about the packet.

One of the bits, EOF, indicates that the current data packet is the last in the frame. Another bit, FID, is toggled when the frame changes. Using these bits, we can detect the beginning and end of a frame. Upon receiving a packet with useful data (not all packets contain useful data), the program copies it to the frame buffer. In my program, I implemented double buffering of the received data (that is, two frame buffers are used).

After the entire frame is received, it needs to be displayed on the screen.

In the case of uncompressed YUY2 data mode, this is quite simple—every 4 bytes of data from the frame buffer correspond to two image pixels.

Note that a YUY2 image measuring 160x120 takes up 38,400 bytes of RAM. In principle, if color information isn't needed for image processing, half the data transmitted by the camera can be discarded, thereby halving the frame buffer size.

With MJPEG, data processing is more complex—each frame is a JPEG image. To decode the resulting images, I used the [TJpgDec](#) library by elm-chan. However, even here, things aren't so simple—I'll simply quote [Wikipedia](#) :

The header of each encoded MJPEG typically complies with the JPEG standard, but some non-compliance is acceptable. For example, it may lack the DHT marker, which specifies the tables for Huffman decoding. In this case, the decoding process should use the tables specified in Section K.3 of the JPEG standard (CCITT Rec. T.81).

Thus, I had to modify the MJPEG decoder to accept pre-prepared DHT data from flash memory.

The resulting performance in YUY2 mode, including on-screen rendering, is:
160x120 – 15 FPS.

Performance in MJPEG mode, including on-screen rendering, is:
160x120 – 30 FPS.
320x240 – 12 FPS.
640x480 – 4 FPS.

Demonstration:

It's worth noting several nuances of using USB video cameras that limit their use with a microcontroller.

- Not all cameras work in UVC mode. I've encountered several such cameras.
- The STM32's built-in USB Host only works in Full Speed (FS) mode. I've encountered cameras that didn't transmit any data at all in this mode. In fact, I connected exactly this camera to the controller first and spent a long time trying to figure out why nothing was working. Later, I connected it to a PC with USB 2.0 disabled in the BIOS and confirmed that the camera also didn't work with it. In HS mode, the camera worked with the PC without any issues.
- The supported resolutions for FS and HS modes differ (the camera transmits different descriptors depending on the mode), and there may be fewer of them in FS mode.
- Resolution options may vary in YUY2 and MJPEG modes.

I also wrote an example of connecting a camera to [the STM32F4-DISCOVERY](#) . Since I didn't have a screen to connect to this board, in this example I simply send the received frame to the PC. This is done using the special IAR debugging mechanism (using yfuns.h).

And finally:

When using YUY2 mode, the received image can be viewed on the PC using the 7yuv program (you need to select the YUV422 YUYV mode).

For analyzing device descriptors connected to the PC, the USBView utility from Microsoft is very convenient.

Just in case, here's [a detailed article](#) about connecting a USB camera to the ARM Cortex-M3 SAM3X.

The project is on Github: github.com/iliasam/STM32_HOST_UVC_Camera

STM32 , USB , Camera , Host

+9

September 29, 2018, 10:18 PM

citizen