ECE342, Lab 5, 2025
University of Toronto

DMA/DCMI using the OV7670 camera

Karthik Ganesan

Introduction

In this lab, you will be interfacing to an external camera model using the Nucleo platform. You will be using the OV7670 module, a board including a camera and configurable DSP functionalities. To interface with the OV7670, you will be using the I^{2}C and DCMI protocols, and the Direct Memory Address (DMA) feature of the Nucleo platform. At the end of this lab document, you will find a list of items that you must show your TA when being marked during your in-lab demo.

1. The OV7670 module

The OV7670 is a versatile camera module that allows you to configure a number of features such as frame size, frame rate, and pixel format, as well as timing, clock polarity, and others. You can configure these settings by writing to the appropriate registers using the camera's propritary SCCB protocol. This protocol is very similar to the I^{2}C protocol you used in lab 4, to connect to the SSD1306 OLED screen.
The complete list of all 209 registers you can set for this camera is available in the OV7670 datasheet. However, for this lab, you will just need a preselected configuration, in which the camera outputs a stream of QCIF frames (which are 144×174 pixels in size) at 15 frames per second (fps), in the $YCbCr$ color format. As you saw in class, the camera module transmits one full image (i.e., one frame) at a time. For each frame, the camera sends each 'row' of the image and for each row, it sends each pixel one after another. Our ultimate goal is to save each frame as one large 'array' in memory. This allows us to then operate on that image (e.g., if we want to post-process the image to detect objects in it or maybe we want to transmit the image wireless over bluetooth etc).
The camera module uses a complex signalling format for sending each frame. For the OV7670, this includes signals such as:

- D0-D7: parallel data lines representing 8 bits of a single pixel.
- Pixel Clock (PCLK): changes every time pixel data changes.
- Horizontal Sync (HSYNC): indicates when one line of the frame has been transmitted.
- Vertical Sync (VSYNC): indicates when an entire frame has been transmitted.

The Nucleo board must read these signals to understand what information is being sent. One option would be to write code ourselves for reading these signals to figure out which row and pixel is being sent each clock cycle and how it should be saved in memory. However, this is **extremely** error-prone and does not make efficient use of the CPU. Fortunately, the Nucleo platform includes a dedicated peripheral called the *Digital Camera Interface* (DCMI) which handles this signalling for us. For this lab, you only need to configure the DCMI to connect to the camera to read images from it.

1.1 Color formats

For this lab, we will only be sending images in 8-bit grayscale. This means that each pixel will consist of 1 byte; `0x00` will represent a black pixel and `0xFF` will represent a white pixel. The OV7670 camera does not support a grayscale mode. We will instead be using the $YCbCr$ mode, where the Y component represents the image 'brightness' and Cb and Cr encode color information.
When reading a pixel from the camera in $YCbCr$ mode, each pixel will be represented as 16-bits. However, to reduce bandwidth the Cb and Cr components are only sent for every second pixel. For example, if Y_i, Cb_i and Cr_i represents the Y, Cb and Cr components for pixel i, the bytes sent from the camera are as follows: $Cb_0, Y_0, Cr_0, Y_1, Cb_2, Y_2, Cr_2, Y_3, Cb_4, Y_4, Cr_4, Y_5$.
If we wanted to represent every pixel with full color, we would need to first re-arrange these bytes to be in the right order. However, we only want to display images in grayscale. This means that you can simply ignore every byte that is not a Y component. **NOTE:** But keep in mind that the camera still sends 16-bits per pixel. You must handle choosing the right byte in your code.

1.2 Set up HCLK

The OV7670 peripheral needs an external clock signal to function properly. We provide this clock from the Nucleo board by assigning one GPIO pin to a Pulse Width Modulated (PWM) signal generated with a timer. The use of a dedicated hardware timer allows the Nucleo board to generate a fast clock without overloading the CPU. For this lab, we use Timer 1 of the Nucleo Board with 8 MHz frequency, and 50 % duty cycle.

Try it yourself: Look at the HCLK signal with an oscilloscope. Do you observe a square wave? Why or why not?

DMA

While the DCMI handles the signalling with the camera, we still need a way to copy the DCMI output to memory. For this we will use the Direct Memory Address (DMA) capability of the Nucleo platform. DMA is a dedicated hardware block which can read and write to memory without requiring the CPU. DMA is particularly helpful when you need to perform a long series of reads and writes, which is exactly what we are trying to do when copying an image from the camera to memory.
In this lab, you will use the DMA functionality to copy image frames captured through DCMI into system memory. When a new frame is available, the DMA interrupt is triggered to copy the frame to memory. The rest of the time, the CPU is free to work on other tasks.

2. Hardware Set Up

First, connect the OV7670 module to the Nucleo board. Using jumper cables, make the connections shown in Figures 1 and 2. The connections are also listed in the Table.
WARNING: Please take your time and make these connections **CAREFULLY**. A very common reason for the camera ‘not working’ is because the connections are not correct. Also, be **VERY CAREFUL** to connect Vdd and GND correctly. If you mix these up, you can permanently damage your camera module and be unable to use it for this lab.

2.1 Sending Images Through Serial Port

When working with images, it is important to have a way to visualize the results. In this lab you will use the serial port to transmit image data from the Nucleo board to your computer, and visualize the images with the `serial_monitor` program provided with this hand out.
First, download `serial_monitor.exe` and make sure you have permissions to execute it. Open a command line interface and run `.\serial_monitor.exe --help`. You should see the following output:

```
> .\serial_monitor.exe --help
Usage: serial_monitor.py [OPTIONS]

  Display images transferred through serial port. Press 'q' to close.

Options:
  -p, --port TEXT           Serial (COM) port of the target board
  -br, --baudrate INTEGER   Serial port baudrate
  --timeout INTEGER         Serial port timeout
  --rows INTEGER            Number of rows in the image
  --cols INTEGER            Number of columns in the image
  --preamble TEXT           Preamble string before the frame
  --suffix TEXT             Suffix string after receiving the frame
  --help                    Show this message and exit.
```

Modify the `main` function in Keil μVision to send the following sequence through serial port:

- String `"\r\nPREAMBLE!\r\n"`.
- The character `0x00` $25,056$ times (144×174).

On your computer, run `.\serial_monitor.exe -p <Nucleo board COM port>`. If your implementation is correct, a graphical window will open showing a black rectangle. Next, modify your code to output character `0xFF` instead of `0x00`. The `serial_monitor` should show a completely white rectangle. You may experiment with shades of gray, or alternating patterns.
When you can reliably send and visualize artificially generated images, you may proceed to the next section.

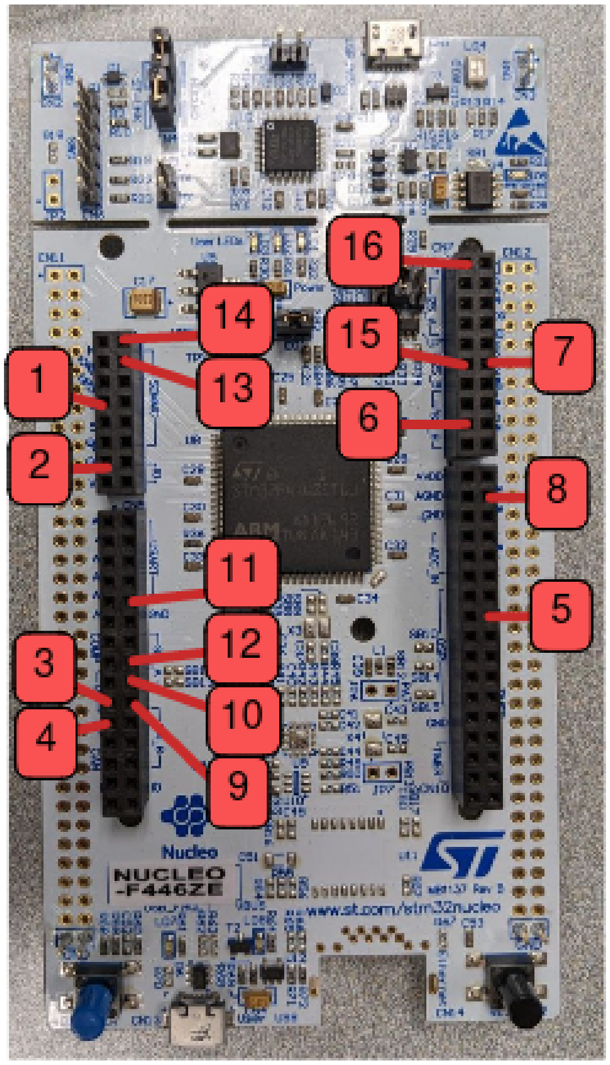

Figure 1

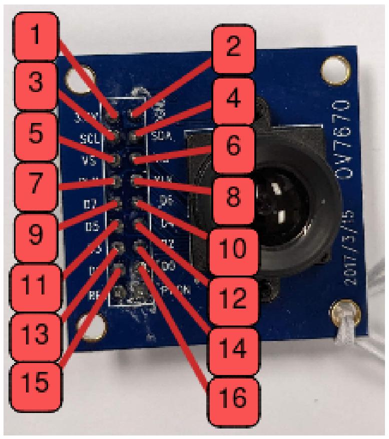

Figure 2

I/O Number	Camera I/O	I/O Label on Nucleo
1	3.3V	3V3
2	DGND	GND
3	SCL	I2C (top)
4	SDA	I2C (bottom)
5	VS	RX DO
6	HS	SPI_B (2nd from bottom)
7	PLK	D12
8	XLK	D6
9	D7	SAI_A (1st from bottom)
10	D6	SAI_A (2nd from bottom)
11	D5	USART (1st from bottom)
12	D4	SAI_A (2nd from top)
13	D3	SDMMC (2nd from top)
14	D2	SDMMC (1st from top)
15	D1	I2S_B (bottom)
16	D0	I2S_A (top)

3. Taking a Snapshot on Command

In this section you will configure the OV7670 camera with I^2C , and use the camera to take a snapshot when button `B1` is pressed. Here are the steps you must take to do this:

3.1 Writing to a register

First, complete functions `ov7670_read()` and `ov7670_write()` in file `ov7670.c`.
Unlike Lab 4, where the RTC module was fully I^2C compatible, the SCCB interface used by the OV7670 is more limited. Thus, the `HAL_I2C_MemRead` and `HAL_I2C_MemWrite` functions will not work as expected here.
You should instead use the `HAL_I2C_Master_Transmit` and `HAL_I2C_Master_Receive` functions. These work very similarly to the functions you used in Lab 4. `HAL_I2C_Master_Transmit` is described as follows:

```
HAL_StatusTypeDef HAL_Master_Transmit(
I2C_HandleTypeDef * hi2c,
uint16_t  DevAddress,
uint8_t    * pData,
uint16_t  Size,
uint32_t  Timeout )
```

`HAL_I2C_Master_Receive` is the same, with the only difference being that `pData` is *sent* in one but *received* in the other.
Both these functions return `HAL_OK` if the operation was successful and `HAL_ERROR` if they fail.
A few things to watch out for, when writing `ov7670_read()` and `ov7670_write()`.

1. Because the controller on the OV7670 is not very fast, you should use a generous timeout (e.g., 10,000) for these functions.
2. Also, they may not work the first time. So you should call them in a loop and only exit the loop when you get `HAL_OK`.

To verify that you can read a register from the camera, try reading register `0x0A` and verify that it contains the correct product number of `0x76`.

3.2 Configure the camera

Next, you must write code to configure the camera at the start. This involves writing to lots of registers that configure the camera to work the way we want. To make this part easier, the `ov7670.c` file contains a large array called `OV7670_reg[]`. This array lists the address of every register in the camera and what value that register needs to be set to. To initialize the camera, you must complete `ov7670_init()` by writing the *value* to the *address* of each register in `OV7670_reg[]`. Since the camera does not run as fast as the CPU, you should leave a 10ms delay between writes (using `HAL_Delay(10)`).

3.3 Configuring the DMA

You must now complete `ov7670_snapshot()` to take one photo using the camera, and then copy that picture (or "frame") to memory. This requires starting the DMA using the `HAL_DCMI_Start_DMA` function. The description of this function is as follows:

```
HAL_StatusTypeDef HAL_DCMI_Start_DMA(
    DCMI_HandleTypeDef * hdcmi,
    uint32_t             DCMI_Mode,
    uint32_t             pData,
    uint32_t             Length )
```

Parameters:

1. `hdcmi` : pointer to a `DCMI_HandleTypeDef` structure that contains the configuration information for DCMI.
2. `DCMI_Mode` : DCMI capture mode snapshot or continuous grab. You can use `DCMI_MODE_SNAPSHOT` for this part.
3. `pData` : The destination memory Buffer address.
4. `Length` : The length of capture to be transferred.

Now, complete function `ov7670_snapshot()` by calling `HAL_DCMI_Start_DMA()` with the appropriate parameters.
NOTE: Notice that `pData` accepts 32-bit values while the camera transmits 16-bits per pixel. So think about how the `Length` parameter will change because of this.
The DMA has been configured so that it fires the `DMA2_Stream1` interrupt when it is done copying one frame. If the `Length` parameter is wrong, the DMA may never complete that many transfers and this interrupt will never trigger.

3.4 Capturing one frame

Now, modify `main()` to capture a snapshot when button `B1` is pressed. You should do this by calling the `0v7670_snapshot()` function you completed above. However, since you are using interrupts, you need a way to tell when the DMA has finished transferring an image. For this, you should modify `DMA2_Stream1_IRQHandler()` in `stm32f4xx_it.c` to set a flag that you can read in `main()`.
You should include a small `HAL_Delay()` inside any loop where you are waiting for the DMA interrupt to trigger. The reason for this is unclear; but not doing this can sometimes cause the DMA interrupt to never trigger. This is likely an issue with the STM codebase.

3.5 Sending one frame

When an image has been captured, you should then send it through the serial port. Remember that the raw image is in *YCbCr* format. To produce a grayscale image you must only send the odd-indexed bytes form the buffer (i.e. `buff[1]`, `buff[3]`, `buff[5]`, etc.).
WARNING: If you always get an image that is almost totally gray with just a few edges visible, you may be sending the *wrong* bytes (`buff[0]`, `buff[2]`, `buff[4]`, etc.).

4. Producing a Video-Like Stream

Once you are able to capture one frame by pressing `B1`, you should modify your code to continuously capture and display images. You should do this using the following steps:

4.1 Using the continuous DCMI mode

Complete function `ov7670_capture()` by calling `HAL_DCMI_Start_DMA()` with the appropriate parameters to capture incoming data on a loop. The only change you need to make is to start the DMA in ‘continuous’ (i.e., `DCMI_MODE_CONTINUOUS`) instead of the ‘snapshot’ (i.e., `DCMI_MODE_SNAPSHOT`) mode you used earlier.

4.2 Modify main to work continuously

You must also modify `main()` to continuously capture snapshots and send them through the serial port. Take care to make your implementation as efficient as possible. Be careful about adding unnecessary delays in your code.

4.3 Avoiding tearing

Your code must run as fast as possible, without ‘tearing’ which occurs when you update the image buffer as you are transmitting it. You can see this as a horizontal ‘tear’ along the image; the part above the tear belongs to one frame and the part below belongs to another frame. One way to avoid screen tearing is by suspending the DCMI hardware when a new frame is captured, and resuming after it has been processed. You can do this by calling the `HAL_DCMI_Suspend()` and `HAL_DCMI_Resume()` functions. Both these functions take a single argument: the pointer to a `DCMI_HandleTypeDef` structure.

Try it yourself: Assuming that the UART communication is the bottleneck, what is the highest frame rate possible?

Try it yourself: Use a timer to profile your implementation. How long does it take to capture a frame and make it available by DMA? How long does it take to send it through serial port?

5. In-lab Demo

During your lab session, you must show your TA the following: send a video-like continuous stream of images through the serial port. Visualize this video using the `serial_monitor` tool. Comment on your results, including how many frames per second you achieved. If your video functionality is working, this is all you need to show your TA.

If your video functionality is not working, you should demo the snapshot mode. For this mode, when `B1` is pressed, capture a snapshot and send it to the computer through serial port. For full marks, your images should show little or no screen tearing.

Your TA may ask questions about your solution. Be prepared to explain your design decisions and your code.