CMOS camera as a sensor

Posted on March 24, 2012, by Ibrahim KAMAL, in Sensor & Measurement, tagged

Today I am going to talk about low cost and effective image processing for very specific embedded applications. I am not talking about robots recognizing their environment or finding their way to a power plug, but rather using small CMOS camera as *better* sensor. We have used this technology for various clients in our consulting service, so I am not going to get into the very specifics of any of those applications cause it would be a breach to our NDAs. Still, IKALOGIC aims to educate and share knowledge to the world. Considering that, I thought about writing a short tutorial, showing to beginners how to get started in that rather intimidating field.

Myths and truthes about image processing

Although there are many tutorials and articles out there on this subject, they tend to discuss the most advanced and complex solutions. wich is – to be honnest – often very impressive, but can be very discouraging if you're just getting started. It's important to say that in many application, all this complexity is not even needed and just a camera like this one **[add reference and link]** and a xMega or ATMega AVR micro controller can do the job just fine.

There are many myths about image processing in embedded applications, here are some of them:

- It will cost a lot of money
- It needs at least an ARM micro controller running quite fast (> 60 MHz) or more powerful
- It needs to be implemented via *MatLab* or other software running on a computer

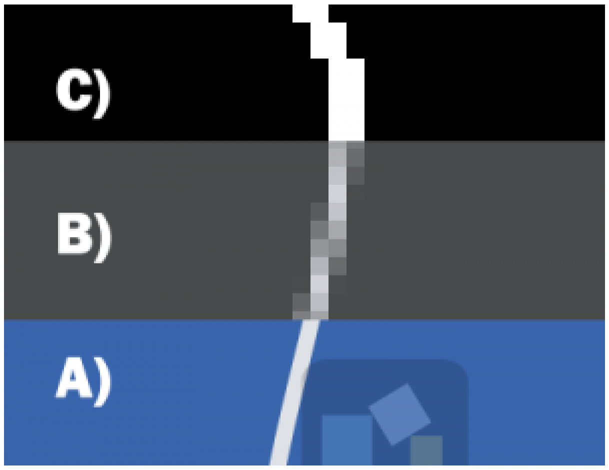
Well, depending on your application, all of the above might be turn out to be false.

Think out of the box

Before we get into the "how" part, let's talk about the "why". I would like you to put aside the idea that using a camera in your project means taking full resolution pictures, and then start applying an Ad hoc algorithms! You need to think about the final aim, the issue you need to resolve. You need to think out of the box and look at the camera as much more than that, or much less! it all depend on your application. To explain my point of view let’s look at this applications:

Robotics competition play ground

Many robotics competitions imply the usage of line sensors, to follow guidance lines traced on the ground plane. Most competitors will build some kind of optical sensors array for this purpose. This solution tends to be the cause of many “bugs” due to the variation of ambient light which leads to faulty sensor readings. Trust me on this, I have been in that exact same place back when I was a student. Some other competitors will completely get rid of the lines traced on the ground plance, and will use very complex camera based navigation where they direct the camera to the front of the robot.Then they and have to deal with the analysis of all other moving objects and elements and implement various algorithms like shape recognition. So, what I’m mean is **why not use the camera as a line sensor array**?



Using a CMOS sensor as a better line sensor

You do not have to use the full resolution and full power of your embedded camera, it can be more interresting to focus the image processing on a specific part of the image. In this case, this would be a couple of horisontal lines. As the picture above shows, you can start with a high resolution image (part A of the picture above), reduce the resolution and get rid of color information (Part B) and simply convert the the pixels into binary information (Part C). This last step is quite important and can be done in many way, but the simplest is to just apply a simple algorithm like that:

```
if Intensity > theshold then
   pixel = 1 (white)
else
   pixel = 0 (Black)
end if
```

By the way, once you “degraded” the image to a couple dozen or hundreds of pixels, processing can become much less demanding an can be done by a cheap 8 bit micro controller.

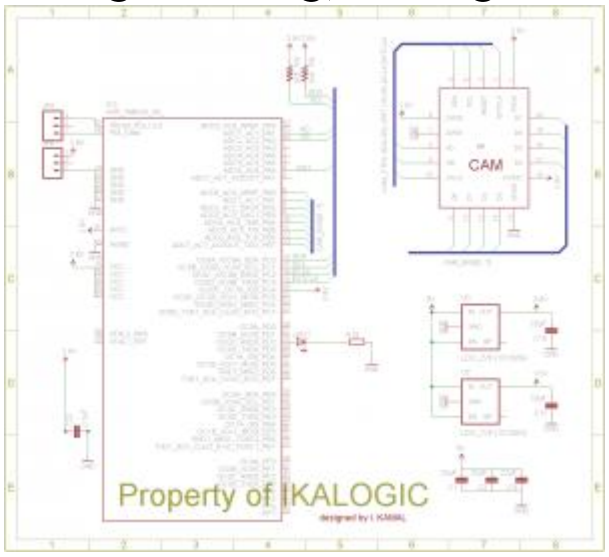
Price? i am sure that a few dollars (< $7) CMOS camera will cost less than an *equivalent* array of LEDs transmitters and receivers, if you count all the PCB space it will occupy.

Another very simple example is using the camera as a proximity switch in rolling belts systems. If you think about it I am sure you can find many situations where a camera can be used as an “enhanced” optical switch. It does not have to imply very complex image post processing algorithms, but the few analysis you can make with the richer information comming from the camera can be enough to solve many problems you faced with a standard optical proximity switch.

I *wish* I could speak of all the ‘unconventional’ applications where we used tiny camera modules to provide reliable solutions to clients. In most of the cases, the client didn’t ever consider using a camera.

If there is one point I want to stress on, is that due to the increasingly dropping components costs, you can now start using embedded cameras and image processing where it does not intially belong. It’s all about degrading the image quality, to increase performance of your system! ironic isn’t it?

Enough talking, let’s get our hands dirty!



Schematic of the TCM8230MD CMOS camera connected to an ATxMega A3

For the sake of our example, we are going to use the **TCM8230MD** CMOS camera, easily available by many distributors like SparkFun. I am going to share an example (working and tested!) schematic diagram showing the electrical connection between the CMOS camera, the micro controller and different power supply sources. Below is the datasheet of the camera we are using:

TCM8230MD

You may think that this datasheet it not very well written, but trust me – in the field of CMOS cameras – this is a fairly well written datasheet! You have all the information needed to understand the function of each signal.

In order to go on with this article and understand the algorithm, I encourage you to study in the datasheet the following signals: EXT CLK, VD (Vertical Sync), HD (Horizontal Sync), DCLK and of course the 8 DOUT signals (D0 to D7).

In the schematic, you will notice that the AtXmega micro controller is providing the clock (EXT CLK) for the camera, this is a key feature of this design, as it will allow us to adapt the data rate of the camera to the capabilities of the micro controller.

Initializing the camera

In this section I am going to explain the most basic interfacing sequence using an AtXmega 256 A3 micro controller. Please note that the following code blocks are not complete functional source codes, but just blocks taken from a bigger complete project.

At some point, usually at system start-up, we need to initialize the camera. There are many ways to do that (a start up sequence is described in the datasheet). Below is an initialization code we have used before:

```
//Camera Initialisation sequence


    CAM_ICLK_OFF;

    CAM_RESET_ON;

    CAM_PWR_ON;

    _delay_us(500);


    CAM_ICLK_ON;

    _delay_us(500);

    CAM_RESET_OFF;

    _delay_ms(500);


    TWI_MasterInit(&twiMaster,

                   &TWIC,

                   TWI_MASTER_INTLVL_LO_gc,

                   TWI_BAUDSETTING);

    cam_sendBuffer[0] = 0x02; cam_sendBuffer[1] = 0x40;

    TWI_MasterWriteRead(&twiMaster,

                        CAM_SLAVE_ADDRESS,

                        &cam_sendBuffer[0],

                        2,

                        0);

    while (twiMaster.status != TWIM_STATUS_READY) {}

    //adjust saturation

    cam_sendBuffer[0] = 0x18; cam_sendBuffer[1] = 0x0;

    TWI_MasterWriteRead(&twiMaster,
                      CAM_SLAVE_ADDRESS,
                      &cam_sendBuffer[0],
                      2,
                      0);
while (twiMaster.status != TWIM_STATUS_READY) {}

//adjust contrast
cam_sendBuffer[0] = 0x11; cam_sendBuffer[1] = init[6];
TWI_MasterWriteRead(&twiMaster,
                      CAM_SLAVE_ADDRESS,
                      &cam_sendBuffer[0],
                      2,
                      0);
while (twiMaster.status != TWIM_STATUS_READY) {}
//adjust brightness
cam_sendBuffer[0] = 0x12; cam_sendBuffer[1] = init[7];
TWI_MasterWriteRead(&twiMaster,
                      CAM_SLAVE_ADDRESS,
                      &cam_sendBuffer[0],
                      2,
                      0);
while (twiMaster.status != TWIM_STATUS_READY) {}
cam_sendBuffer[0] = 0x03; cam_sendBuffer[1] = 0x22;
TWI_MasterWriteRead(&twiMaster,
                      CAM_SLAVE_ADDRESS,
                      &cam_sendBuffer[0],
                      2,
                      0);
while (twiMaster.status != TWIM_STATUS_READY) {}
```

What this piece of code does is :

- Start the internal clock (generated by a timer on pin PC3). the line “CAM_ICLK_ON” is simply a macro that can be replaced by “PORTC_DIR |= (1<<3); “. In other words, PC3 is configured to output a clock via a timer (around 12MHz) and we simply set PC3 as output when want to “output” this clock.
- Setup the I2C port – nothing special about that except that ATMEL calls it TWI (Two Wire Interface)
- Send a series of commands to adjust the CMOS camera operation. each command is composed of two bytes. Please refer to the datasheet for the meaning of each of those commands.
  After the execution of this code, and if a clock is provided to the camera, it will output pixel information all synchronized with VD, HD and DCLK signals.

Taking a picture

Now when you have the signals coming out from the camera you *will* face a quite frustrating problem! You will want to store the pixel information into a 2D array. No matter how well written your code is it will be complicated to synchronize the time you read the pixels with the 3 signals that define the X and Y coordinate of that pixels. The problem is that if you want to take pictures fast enough you won’t be able to follow up with the clock , and if you slow down the clock, it will take too much time to “scan” the CMOS sensor and the image will be somehow blurry or distorted. **Think of the time taken to clock out all the pixels of a frame as the shutter speed.**

The idea we came up with to overcome this problem is to “stop” or slow down the clock from time to time when the micro controller can’t keep up. simple, really simple. So, here is the code we use to take a picture:

```
loop_until_bit_is_clear(VPORT0_IN,CAM_VD);     //wait for a

loop_until_bit_is_set(VPORT0_IN,CAM_VD);     //new frame

TCC0.CCA = 1;     //divide the clock by 2

                //normally TCC0.CCA = 0

for (y=0; y < 96; y++)

{

        asm volatile(

        "cli"                    "\n\t"

        "L1: in r24, 0x0012"     "\n\t" //read portA_in (mapped to port0) to
r24

        "sbrs r24,1"             "\n\t" // wait for the rising edge on cam HD
(new line)

        "rjmp L1"                "\n\t" //loop
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"

        "L2:nop"                 "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "nop"                     "\n\t"
        "in r24, 0x0016"         "\n\t"    //read pixel info from portB_in
(mapped to port1) to r24
        "st %a0+, r24"            "\n\t"
        "in r24, 0x0012"        "\n\t" //read portA_in to r24

        "sbrc r24,1"            "\n\t" // wait for the falling edge on cam HD
(end of line)

        "rjmp L2"                "\n\t" //loop

        "sei"                   "\n\t" //Done, turn

        :

        : "e" (frame[y])

        : "r24"

    );

    TCC0.CCA = 0; //go back to fast clock

}
```

The code above simply waits for a rising edge on VD signal then drop the clock speed a little. Then it will loop through the 96 lines of the image (we were using 128*96 image format). At each loop it will store the 128 pixels of the line. Note that for this code, we were only interested in the RED component so we only read one pixel via port B.

As explained in the code comments, we used virtual ports 0 and 1 to access port A and port B respectively. Virtual ports exist only in the Xmega family of the AVRs and allow to use the fast asm “In” and “Out” commands which execute in 1 clock cycle only.

At this point, it’s up to you to work your way for your specific application.

That’s it for this tutorial. As I said before, it’s important that you read the datasheet before getting into those code snippets.

Below I have included an eagle project of the schematic above, it has the pcb foot print of all the components including the TCM8230MD CMOS camera.

Source: http://www.ikalogic.com/image-processing-as-a-sensor/