Peeping ROM: A Digital Camera System for Students and Educators

Sean Muller, Michael Cirelli, Jim Paris, Andrew Muth

10 August 2005

Contents

The Peeping ROM Quick Start Guide

This Quick Start guide is provided to eliminate many of the common problems experienced when setting up Peeping ROM for the first time.

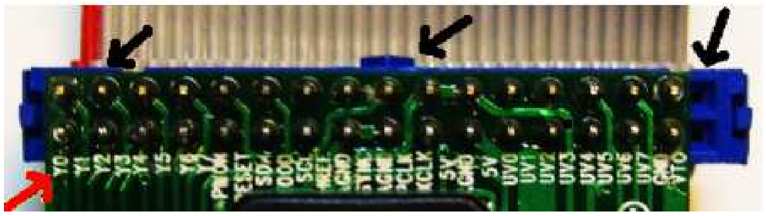
- If you are setting up Peeping ROM for the first time, use this quickstart guide to get your system connected and tested.
- If you want detailed explanations and usage examples for each of the major Matlab functions that run Peeping ROM, please see section 3 on page 12.
- Experienced users looking for detailed schematics and code listings can find information in appendicies A and B on pages 22 and 47, respectively.
- Finally, those looking for information on the experimental work that led to a model of Peeping ROM as a vintage film camera should consult section 2 on page 6.

Hardware Setup

Please pay careful attention to the hardware setup instructions as it is possible to break the camera module by connecting it improperly.

Hardware setup involves connecting the camera and printed circuit board together with a 34-pin ribbon cable. When you receive your Peeping ROM hardware these components should already be connected; however they are not permanently attached and may need to be reconnected if they become seperated.

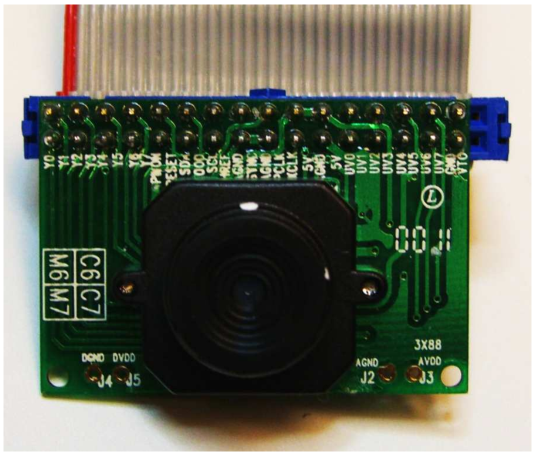
On the circuit board a polarized hood forces the cable into its proper orientation, but on the camera there is no similar safeguard. To attach the cable to the camera, **place the header into the cable with the text `Y0` on the side with red indicator wire.** It should look like this:



Header Connection Close-up

Note that the polarizing "key" of the cable connector is **still visible** and the two unused holes of the connector are **opposite the red indicator wire**. Also check that all pins are **within the connector housing** since the camera can be connected with only some pins inserted into the cable.

If properly connected, your camera module should look like this:



Camera/Cable Connection

If it does not look exactly like the picture, **DO NOT CONNECTOR POWER** to your Peeping ROM board. You will break the module. Only after triple checking that this connection is correct should you connect a serial cable between the PC and your hardware board. Finally, connect $+8 - 35$V to the main circuit board to provide power. If a bright green indicator light does not come on immediately, disconnect power and check your camera connection.

Software Setup

Peeping ROM's Matlab code is designed to run on a variety of operating systems and Matlab versions. It runs correctly under Matlab v6.5 and v7.01 on both Linux and Windows.

To operate Peeping ROM, you should open Matlab then change your current directory to the one containing all of the files listed on page 47. They contain all of the necessary functions to communicate with the camera module and download an image.

Testing Your System

Testing your camera setup can be done using the Matlab function `test_me`. Once you have completed the hardware and software setup, make sure the green indicator light is glowing on the Peeping ROM board, then at the Matlab prompt type `test_me`. If everything is correct, you should see a bunch of text appear on the screen as internal functions initialize communications, check that the camera is working, and read the value of several camera parameters. Finally, after about 30 seconds of downloading, your photograph should appear.

Note: it is likely that the images you see after calling `test_me` will be blurry – this is not an error from Peeping ROM. The camera lens simply needs to be focused.

Note: the proper orientation for the camera is to have the header rows and white focusing marks at the top. Images will appear right-side-up if the camera is held this way.

Focusing the Camera

When Peeping ROM's camera is used with its factory-shipped lens, focusing the unit for a particular setting is more of an art than a science. Nevertheless, every Peeping ROM camera comes with two white "focusing" marks (one on the lens and one on the camera housing) to make fine adjustments to the camera's focus easier.

Actual focusing requires taking many pictures of a scene and making adjustments as necessary. The easiest way to do this is with `test_me`, though this takes a long time since each photograph takes about a minute to fully process. Experienced users may find the best method to be taking a single `test_me` picture, recording the final register values from the display output, then running `picture_me` with no delay and all of the necessary register values preprogrammed.

1 Introduction

For many people – students, professionals, and laypeople alike – the digital camera is a tool that allows anyone to become a photographer overnight. Pictures are instantly viewable and can be deleted if they are not satisfactory. The time-consuming and expensive process of getting film processed has been done away with completely. Digital storage mediums can store thousands of photos which are easily trasferred to PCs and the internet. However, these very attributes cause many to think of the digital camera as a "magical point-and-shoot box" whose only connection to the film camera is that they both take pictures.

We intend to dispell that myth by exposing you to the digital camera at a whole new level. Forget the "point-and-shoot" box – Peeping ROM takes modern camera technology back to its roots and shows that there are indeed parallels between the inner workings of digital cameras and those of film cameras.

In this report we will accomplish three tasks: explaining the functionality of Peeping ROM's camera in terms of an analogy with the Graphlex *Speedgraphic* camera, providing a full recount of our experimental methods that led to the *Speedgraphic* model of Peeping ROM, and describing in detail everything needed to use our hardware and software framework when performing more tests. We will also provide all code, schematics, and other data relevant to the OmniVision OV6630 module as a reference.

2 Understanding Peeping ROM

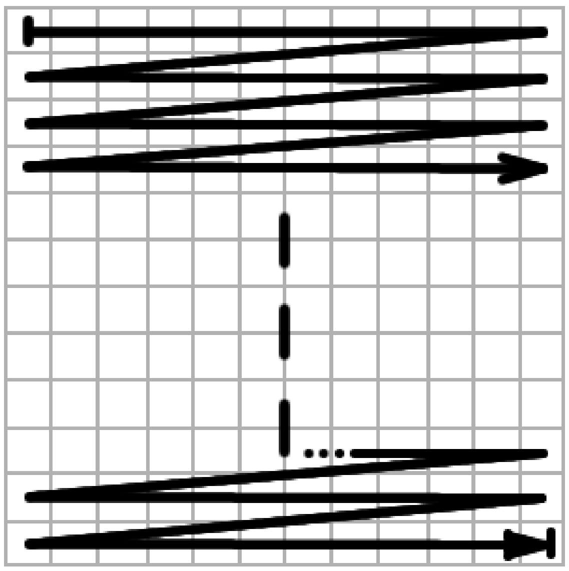
The Focal Plane Shutter & the Graphlex *Speedgraphic*

2.1 First Impressions

At first glance, the technology in Peeping ROM's imager is radically different from that of a film camera. Instead of f-stop, shutter speed, and film speed, there are only electronic controls with names like `GAIN` (system gain), `CLKRC` (clock rate control), `AEC` (autoexposure control), and `COMH` (common control 'H'). These controls, called ***registers***, are bytes of memory that can be set and read electronically. They can contain values from 0 to 255 and may be multiplexed so that specific parts of each value has different effects. See the OV6630 datasheet on page 101 for more information on these camera registers. The lens itself is of a simple fixed-aperture "barrel" type that can be screwed closer to the sensor array to change the focus, but aside from this there are no moving parts in the entire camera. There is no physical shutter or iris. Indeed, when powered up the module continually streams out image frames like a video camera rather than waiting for a triggering signal to capture a single frame.

2.2 The *Speedgraphic* Model

Given the vast differences in functionality noted above, it is apparent that a model is needed to see Peeping ROM's many registers in terms of standard camera controls. In more advanced digital cameras, the CCD's pixels are charged up for a time to make them receptive to light, then exposed by firing a mechanical shutter. Once this shutter has closed, the camera samples each pixel to measure its value and thus figure out how much light fell on it. However, cheaper digital video cameras like webcams lack this mechanical shutter and must make up for it by charging up pixels, waiting, and reading them sequentially while moving down the sensor array.



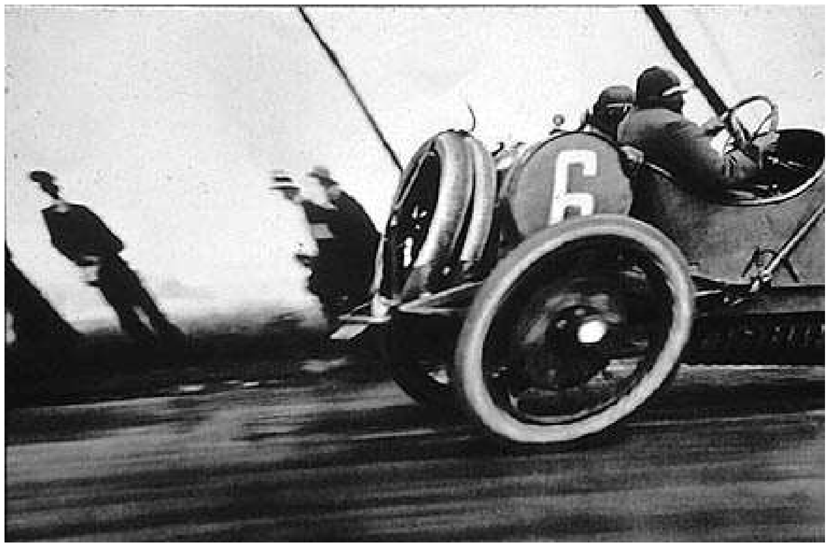
CCD Pixel Reading Pattern & *Graphlex* Speedgraphic *Vertical Shutter*

Based on this knowledge and a basic understanding of the data output from Peeping ROM, we hypothesized that Peeping ROM may act like the Graphlex *Speedgraphic*, a vintage camera that uses a focal plane slit shutter moving vertically from top to bottom. (A focal plane shutter is a shutter that exists in the focal plane of the image rather than in the lens itself.) In these cameras the focal plane shutter has two adjustments: shutter tension select and slit width. The shutter tension selector changes the speed of the shutter cloth (and thus the light-emitting slit) as it moves down the film. The slit width setting, on the other hand, changes the width of the slit itself. Together these controls set the total exposure time (in n^{ths} of a second) and allow the camera to operate in many different lighting environments.

Peeping ROM, while lacking any of the physical controls present on the *Speedgraphic*, has electronic controls that can change the pixel precharging time and overall framerate. We hypothesized that changing these registers may have the same effect as changing shutter speed and slit width. With this hypothesis, we then took more than 50 photographs of a single scene while making various adjustments to the module's registers. These photographs would then provide information about the camera's functionality, design, and potential as a useful tool for showing students the inner workings of a digital camera.

2.3 Testing Peeping ROM

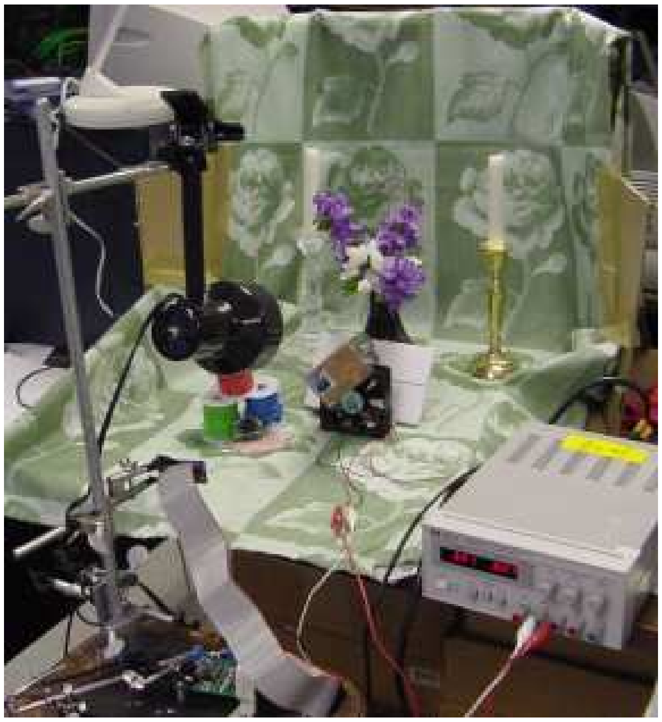
One of the more unique examples of how photographs provide insight into the inner workings of the camera comes from a famous picture taken by Henri Lartigue. Shown below, the image's components "lean" in multiple directions because the camera used a vertically-travelling slit shutter to view moving objects.



Lartigue's 'Leaning Car Race'

In the case of this photograph, the camera is panning right to follow the speeding car – this is evident because the people are leaning left, caused by the camera moving an incremental amount right for each small distance the shutter moved down. Meanwhile, the car tires are seen as right-pointed ovals because the camera is panning faster than the car is moving and thus the car is seeming to move slowly into the center of the frame as the exposure goes on.

Since the "leaning" behavior discussed above is intrinsic to any camera with a vertically-travelling slit shutter, we hoped to validate our hypothesized model of Peeping ROM by photographing a scene with similar moving objects. If the photographs showed similar features we could then have a grasp on the actual electronic shutter mechanism running Peeping ROM. With that knowledge we could then determine which registers, if any, would allow us to vary the slit width and speed of exposure to change the camera's functionality.



Experimental Setup

Our experimental setup was designed to hold the Peeping ROM camera module in one position so multiple pictures could be taken and compared usefully. We attached a card with text and images on it to a small gearmotor whose speed was controlled and monitored using a strobe. The background was chosen to be rich in colors and patterns while spools of red, green, and blue wire provided color references.

We conducted six experimental runs to clarify our model of Peeping ROM:

1. **Gathering Baseline Photographs**
   Testing the camera would require several baseline photographs for comparison. Also, we wanted to get the camera module focused on our scene and the ambient lighting set to acceptable levels.

2. **Enabling and Disabling Automatic Modes**
   To make Peeping ROM useful it is necessary to disable all of the automatic modes that operate the camera normally. However, the methodology used to change these modes is not specified in the datasheet and many of the register settings involved are interconnected. The goal of this set of photos was to figure out the precise register settings needed to deactivate these automatic modes and write a program that could set them according to a user's specification.

3. **Changing Picture Framerates**
   A key component of our Peeping ROM model is the ability to change the apparent shutter speed of our camera. For this reason we chose to explore the effect register `CLKRC` (`0x11`) has on an image. The lower six bits of this register control the clock prescaler, with a higher value dividing the clock by a greater number and thus increasing the expose time. Based on our hypothesis, we would expect a faster clockspeed to act similar to a faster shutter speed, decreasing the amount of light hitting the sensor.

4. **Changing Pixel Precharging Time**
   A second key component of our model is the ability to change the width of the imaginary "slit" in our electronic *Speedgraphic*. Our hypothesis suggested that the pixel precharging time would cause effects similar to widening or shortening the slit. According to the datasheet and our observations of register values during tests 1 and 2, register `AEC` (`0x10`) may have this effect. We expected that lowering this value would cause the picture to darken since each pixel would be exposed to the light source for less time between precharging and measurement.

5. **Recreating Lartigue's "Leaning Car Race"**
   The true test of our vertical focal plane shutter model would be to recreate similar leaning and blurring features in an image from Peeping ROM. Our hypothesis suggested that a long frame rate coupled with the shortest possible exposure time should yield a decent image, albeit one that is on a time scale far slower than Lartigue's.

Motion Apparatus (gearmotor and card)

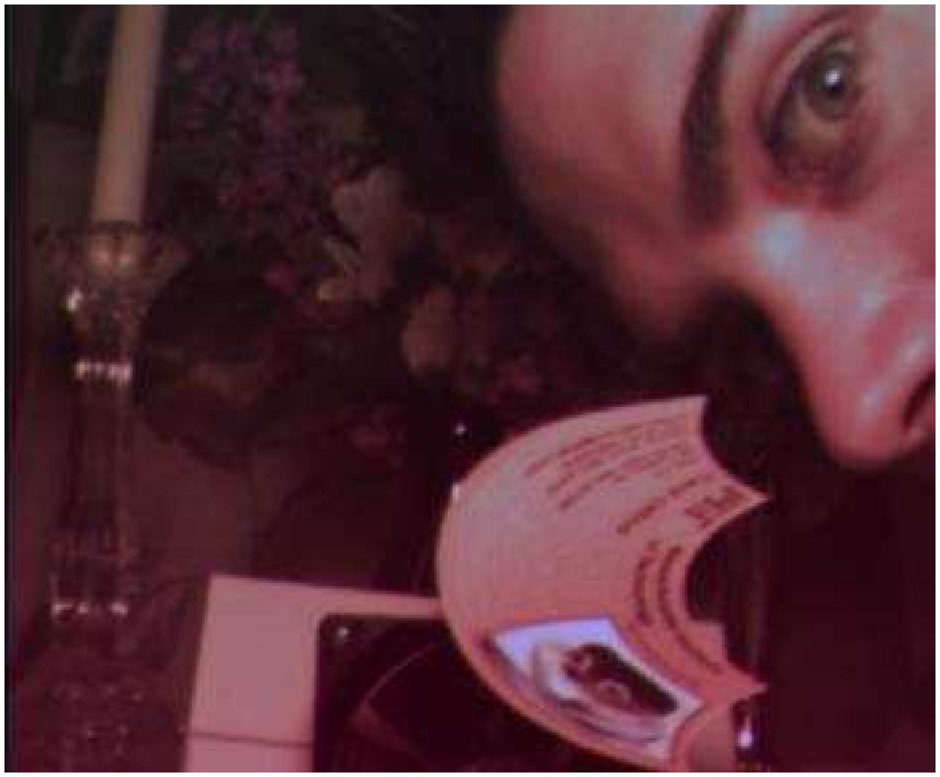
6. **Changing System Gain**
   Controlling the overall gain of Peeping ROM is an exceedingly useful tool to use when using unusual frame rates and exposure timings since many images will be either underexposed or showing blooming. These photographs were designed simply to check that the system gain could be controlled while the automatic gain control is shut off.

These six tests provided us with enough information to validate our Peeping ROM model and actually specify what registers roughly corresponded to physical adjustments on a traditonal film camera. Our conclusions and finalized model are presented below, while details of all of our experimental runs are shown in section 3.

2.4 Conclusions

Overall, the six test series listed earlier gave positive results. `automode` proved useful in getting baseline photos and experimenting with the camera given a variety of lighting environments. After some initial shortcomings we were also able to develop the list of registers that enable the automatic gain/exposure/white balance circuitry, which led directly to our writing of automatic mode configuration function `pr_modeset`.

Once the automatic modes could be disabled without causing unintended effects on the images, we were able to proceed with our exposure control experiments. These produced the expected results with faster framerates and exposure timings (or, using our *Speedgraphic* model, shutter speeds and slit widths) producing pictures with Lartigue's "leaning" appearance. Changing the exposure speed to make very long exposures also allowed for use in low-light situations, although in these cases the automatic gain needed to boost the image often produced a lot of noise in the final photo.



Peeping ROM's "Leaning ID Card with Face"

Of course, the ability to work with the camera in detail also showed that it was at heart still just a cheap webcam. During our experiments with the automatic gain and exposure modes we often found that our images would be colored incorrectly or have a "washed-out" appearance due to reflective materials in the photograph. Freezing motion, one of our largest hopes for the camera once the exposure timing was figured out, was impossible because of the *Speedgraphic*-style shutter. Furthermore, even with the ability to change every register in the camera itself only a very few have any effect on the image itself and of those even fewer have effects that could be called useful.

The following table shows all of the registers we experimented with during our trials and their effect on the camera.

register address	**register name**	**register description**
`0x00`	`GAIN`	Controls the overall system gain. Can be set manually if AGC is disabled.
`0x01`	`BLUE`	Controls gain of the blue channel. Can be manually set if AGC is disabled. Also useful during 10 second "register settling" delay as an indication of image exposure.
`0x02`	`RED`	Controls gain of the red channel. Can be manually set if AGC is disabled. Also useful during 10 second "register settling" delay as an indication of image exposure.
`0x06`	`BRT`	Controls the image brightness. Can be set manually if AWB is disabled.
`0x10`	`AEC`	Controls the pixel exposure delay – time between pixel precharge finish and readout by ADC. Comparable to **slit width** in *Speedgraphic* model. Can be set manually if AEC is disabled.
`0x11`	`CLKRC`	`CLKRC[5:0]` control the frame rate of camera. Adjustable between 44.27 Hz (`CLKRC` = 0, 0.023s exposure) to ∼0.5 Hz (`CLKRC` = 63, ∼2s exposure). Comparable to **shutter speed** in *Speedgraphic* model.
`0x12`	`COMA`	`COMA[1]` enables color bar test pattern. Not useful for experimental purposes.
		`COMA[2]` enables auto white balance system. Startup value is 1 (enabled); `pr_modeset` sets to 0 if given the argument pair `'AWB','disabled'`.
		`COMA[5]` enables auto gain control system. Startup value is 1 (enabled); `pr_modeset` sets to 0 if given the argument pair `'AGC','disabled'`. Also see `COMB[0]` and `COMH[3]`.
`0x13`	`COMB`	`COMB[0]` controls the ability of automatic modes to adjust respective registers. Does **not** disable automatic modes; only prevents them from changing control registers. Will influence multiple automatic modes.
`0x28`	`COMH`	`COMH[3]` disables the automatic gain controls. Startup value is 0 (enabled); `pr_modeset` sets to 1 if given the argument pair `'AGC','disabled'`. Also see `COMA[5]` and `COMB[0]`.
`0x29`	`COMI`	`COMI[7]` disables the automatic exposure controls. Startup value is 0 (enabled); `pr_modeset` sets to 1 if given the argument pair `'AEC','disabled'`. Also see `COMB[0]`.

3 Using the Peeping ROM System

Peeping ROM is designed to make experimenting with the many possibilities of this digital camera system as simple as possible. Every register can be altered using our Matlab scripts, though preprogrammed hardware interlocks are in place to ensure that one cannot tamper with system critical settings. We have also provided several wrapper functions that simplify testing your camera setup, taking a picture, viewing all of the image data, and inputting user changes.

Note: We recommend using at least Matlab v7.0 to run Peeping ROM. While all of the control functions should work with older versions of Matlab, they may take substantially longer (∼300%) to download an image.

3.1 Testing Your Setup with `test_me`

`test_me()` is a function that allows the user to test his or her camera system. It places the camera into all of its automatic modes, waits the proper amount of time, captures an image, and displays it. Though the user cannot specify any register changes to alter the camera settings when using `test_me`, it is useful for capturing baseline photos and viewing what "acceptable" register values are, given a particular lighting environment.

To use this mode, simply ensure that your camera is connected and powered up then call `test_me` at the matlab prompt. After initializing the camera, the program will wait several seconds before capturing a photo and downloading it to the screen. During this delay the program will return data on 7 registers: `BLUE`, `RED`, `CTR`, `BRT`, `SHP`, `AEC`, and `GAIN`.

As a reference, the code listing for `test_me.m` is in article 1 in appendix B.

`automode` *photograph*

3.2 Using `picture_me` to Take Pictures with User Modifications

`picture_me()` is a wrapper function that calls a number of other matlab functions to take a picture *after applying user-specifed changes* to camera registers. More specifically, `picture_me` allocates variables, calls the image capture functions, applies user changes, displays the resulting data, and returns the raw image data in matrix format.

To use this mode, ensure that your camera is connected and powered up then call `picture_me()` at the Matlab prompt. The program will then call the `camera_setup`, `camera_usermode`, `camera_takepic`, and image generation functions (`bayer.m` and `pr_showpics.m`) described in sections 3.2.1 to 3.2.3 to take a photo. All user-specified changes are made in `camera_usermode`.

Note: In general, the user should not need to edit anything in `picture_me` itself; however if a specific serial port is desired it can be entered as an argument to `camera_setup`, *i.e.* `camera_setup('/dev/ttyS2')` or `camera_setup('COM3')`. To alter registers, use the `pr_regset` function in `camera_usermode.m`.

As a reference, the code listing for `picture_me.m` is in article 2 in appendix B. The following sections 3.2.1 through 3.2.3 will describe in detail exactly how the `picture_me` code works, how to interact with it or add user modifications, and how to interpret its graphical output.

3.2.1 camera_setup.m

The purpose of `camera_setup()` is to properly open a serial port and reinitialize the camera system. The program allows for the user to specify a serial port to use; alternatively the program will default to the standard system serial port. For more information on this, see `pr_init.m` in article 13 of appendix B or "Using picture_me to Take Pictures with User Modifications" in section 3.2.

As a reference, the code listing for `camera_setup.m` is in article 3 of appendix B.

3.2.2 camera_usermode.m

`camera_usermode()` is the main function for camera-user interaction and modification. It allows for disabling automatic modes, performing any register changes desired by the user, and displaying a verbose output during the 10-second waiting period before the snapshot occurs.
Interacting with the camera registers can be done in 4 ways:

1. **Enabling or disabling automatic modes with** `pr_modeset`. `pr_modeset` takes six (optionally seven) strings as function arguments, with valid options being `'AEC'` (autoexposure), `'AWB'` (auto white balance), `'AGC'` (auto gain control), `'enabled'`, and `'disabled'`. The function expects arguments in pairs of the form <*`mode`*>,<*`state`*>, though the pairs themselves can occur in any order. As a final argument, `pr_modeset` can take either `'verbose'` or `'quiet'`, though it defaults to quiet mode if no argument is given.

   For example, to turn off the Auto White Balance and see all of the register changes, you would change the `pr_modeset(...)` command (line 10 in `camera_usermode.m`) to read:

   ```
   pr_modeset('AEC','enabled','AGC','enabled','AWB','disabled','verbose');
   ```

   On the other hand, if you wanted to turn all of the automatic modes off and not see any of the register changes on screen, you would enter:

   ```
   pr_modeset('AEC','disabled','AWB','disabled','AGC','disabled','quiet'); or
   pr_modeset('AGC','disabled','AEC','disabled','AWB','disabled');
   ```

   For more information and code listings of `pr_modeset`, please see article 8 in appendix B.

2. **Changing register settings with** `pr_regset`. `pr_regset` takes a list of two (optionally three) arguments to set any of Peeping ROM's 92 registers to a particular value. The function calls should use the following formatting: <*`register number`*>,<*`new value`*>, (<`'verbose'`/`'quiet'`>). These inputs can accept either hexadecimal values formatted as character strings (*i.e.*, `'D3'` or

`'47'`) or decimal values from 0 to 255. If desired, `'verbose'` or `'quiet'` can be passed to see the register being changed, its original value, the value it is being given, and its value after being read again. This feature can be useful as some registers have masks in hardware or strange behavior that prevents them from taking a user-specified value. Like `pr_modeset`, this function defaults to quiet mode if not told otherwise.

To use this function to set register `0x1B` to 128 with the debug output, you would enter:

```
pr_regset('1B',128,'verbose');
```

If you wanted to set register 37 to value `0xFD` with no screen output, you would enter:

```
pr_regset(37,'FD');
```

For more information and code listings of `pr_regset`, please see article 16 in appendix B.

3. **Changing specific bits in a register with `pr_bitset`.** Many of the registers on Peeping ROM's camera are multiplexed such that specific bits have independent functions. For this reason it may be necessary to toggle the state of a particular bit without changing the value of its neighbors. While it is possible to accomplish this with logical operations and register-wide reads and writes, we have provided the function `pr_bitset` to simplify this problem.

   `pr_bitset` expects three (optionally four) arguments of the form <*register number*>,<*bit number*>,<*value*>,(<`'verbose'`/`'quiet'`>). The register input can be either a character string containing a hexadecimal value or a plain decimal input from 0 to 255. The bit argument is *zero-centered* so it ranges from bit 0 (the LSB) to bit 7 (the MSB). The value argument should be either 0 or 1. Finally, this function can accept a fourth `'verbose'` or `'quiet'` argument and defaults to quiet mode if not specified.

   For example, to set bit 4 of register `0x2D` to 1 with a verbose output, you would enter:

```
pr_bitset('2D',4,1,'verbose');
```

   For more information and code listings of `pr_bitset`, please see article 6 in appendix B.

4. **Reading register values with `pr_regread`.** There may be cases when it is desirable to know the value of a register at startup without changing its value. To do this, enter the following line of code (with <*reg#*> set to the value of the register address):

```
pr_regread(reg#);
```

   This will print the value of the register *reg#* in the Matlab terminal. If it is given a character string (*i.e.* `'F3'`) the function will interpret this as a hexadecimal value; otherwise it will expect a decimal value.

   Note: This function uses `sprintf`, an intrinsic Matlab function. For more information on its usage please see Matlab's help files. For more information on `pr_get`, please see article 14 of appendix B.

To perform the changes shown above, `camera_usermode.m` must be edited manually before running `picture_me`. Changing `pr_modeset` is done by directly altering its existing function call on line 10. Adding register or bitset operations should be done after line 18, in place of the comment "`%% insert changes here.`

Note: Changing the output display during the register "settling" period. Despite the fast processor clock on Peeping ROM's camera, the automatic gain/white balance/exposure circuitry still takes time to arrive at an optimal value. During our experimentation we found that it took between 5 and 10 seconds for the camera to reach its optimal values. We also found that it was useful to see the value of several registers on the camera (typically those associated with the automatic modes and exposure timings) over the course of this delay period since they provide insight to the camera's light environment and image output.

If desired, this output can be enabled by changing the `camera_usermode()` call in `picture_me.m` to have the argument `'verbose'`. This will cause the display to show the time remaining before image caputure, as well as the state of registers `BLUE`, `RED`, `CTR`, `BRT`, `SHP`, `AEC`, and `GAIN`, once per second. This mode is turned off by default and must be manually enabled in `picture_me.m`.

Note: It is also possible to remove this 10 second delay completely by sending `camera_usermode` the argument `'no-delay'`. This can speed up the total time it takes to get a picture (it removes 10 seconds of delay), but only use this argument if you have disabled all of the automatic modes **and** set their registers to values that will produce a desirable image. Otherwise, you may not see a useful image.

As a reference, the code listing for `camera_usermode.m` is in article 4 of appendix B.

3.2.3 camera_takepic.m

`camera_takepic` is a wrapper function that tells the camera board to actually perform an image capture, download the image, close the serial port, and return a 349 × 288 matrix containing the downloaded image data. Each byte will be a value from 0 to 255. While the data downloads the percentage complete will display every 5 seconds, and at the end of the download the total time in seconds is shown.

As a reference, the code listing for `camera_takepic.m` is in article 5 of appendix B.

3.3 Image Generation

Image generation in Peeping ROM takes two distinct steps – interpolation of raw data, then the actual on-screen figure generation. Interpolation is done with the bayer pattern decoder function `bayer.m`, which processes the raw data and generates four output matrices corresponding to red, green, blue, and full-color images. For more information on the Bayer pattern and Peeping ROM's interpolation technique, see section 3.3.

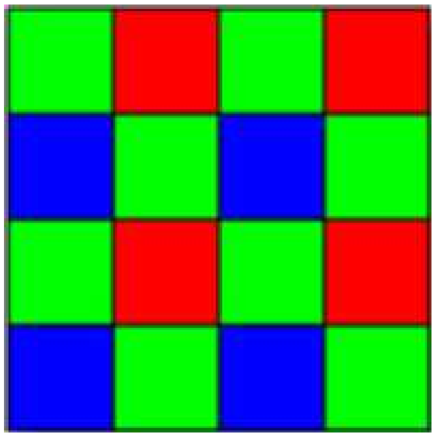
Generating the actual picture output is handled by another function, `pr_showpics`, which takes the output of the bayer interpolation function and displays all of the data. It will display the raw BW image before any interpolation, the red/green/blue pixel data before interpolation, the red/green/blue channel information post-interpolation, a composite brightness histogram, and the final full-color image itself. Each of these images will appear in their own unique frame and will refresh in those identical frames if another image is taken. For more information on interpreting these visual outputs, please see section 3.5.

As a reference, the code listing for `pr_showpics.m` is in article 22 of appendix B.

3.4 Understanding Peeping ROM's Bayer Pattern Interpolation

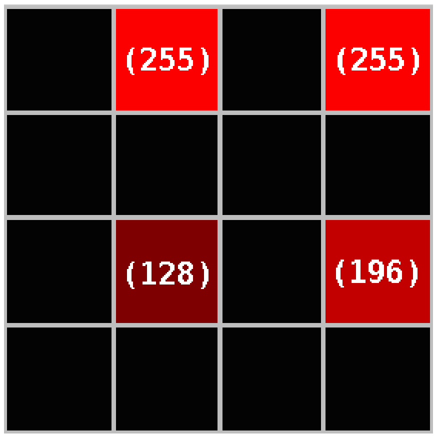
A key difference between the operation of a typical film camera and the Peeping ROM image sensor is that the raw image captured by the camera does not contain the full color information at every pixel. Rather, each pixel is just a small grayscale light intensity meter producing a value from 0 (pure dark) to 255 (pure light). Color sensitivity is achieved using microscopic filters placed over every pixel, thus making each pixel sensitive to only one color. For Peeping ROM, the sensor elements form a 349 × 288 grid with either a red, green, or blue filter covering each pixel in a special arrangement called the Bayer

Pattern. As shown in the following picture, this pattern is an alternating grid with twice as many green points as red or blue points – this accounts for the human eye's hightened sensitivity to green light.



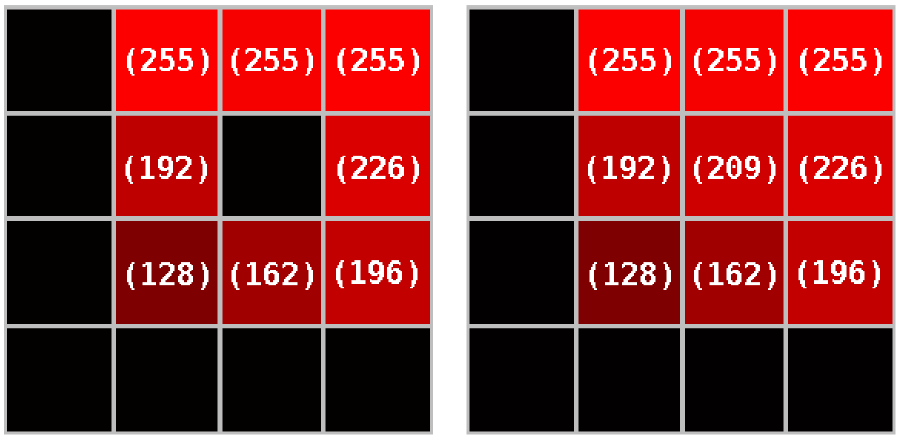
Bayer Pattern

Given the Bayer color filter, an interpolation technique is needed to convert the single-channel input into a full-color, three-channel input. Peeping ROM uses a simple averaging interpolation algorithm since the number of input pixels is the desired number of output pixels and image quality is not of huge importance. In `bayer.m`, the original data input is converted into three matrices – red, green, blue – with each having a value at a particular pixel only if that pixel was covered with the respective matrice's color. All other pixels are set to 0, which leaves them black.



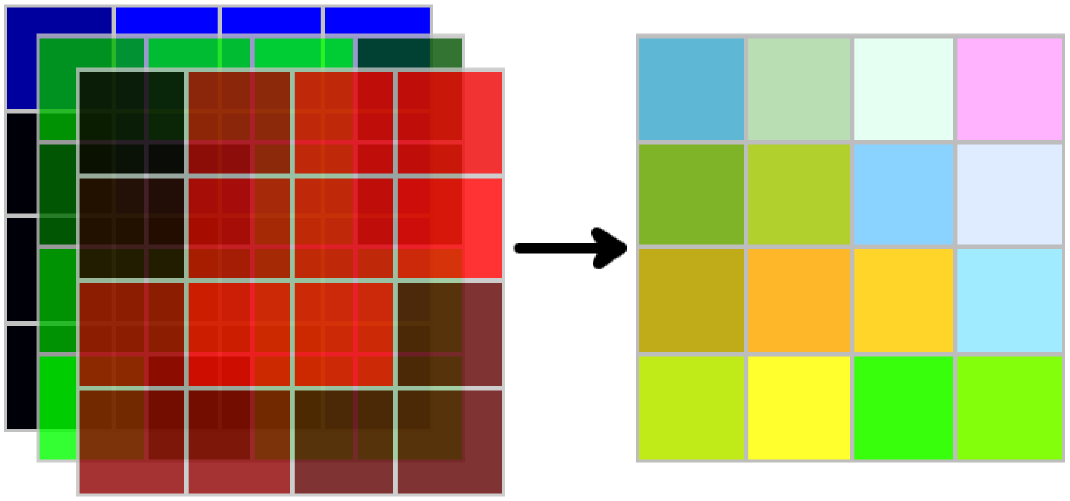
Step 1: Select Only One Color

Next, the closest neighboring pixels are averaged to provide an interpolated value for each pixel that exists in the full image but was covered with a different colored filter.



Step 2: Interpolate Simple and Difficult Cases

The actual averaging may include simple two-element systems or more complex four-element systems. The green channel contains only four-element systems since the pixel arrangement is different compared to either the red or blue channels.



Step 3: Create Multicolor Image

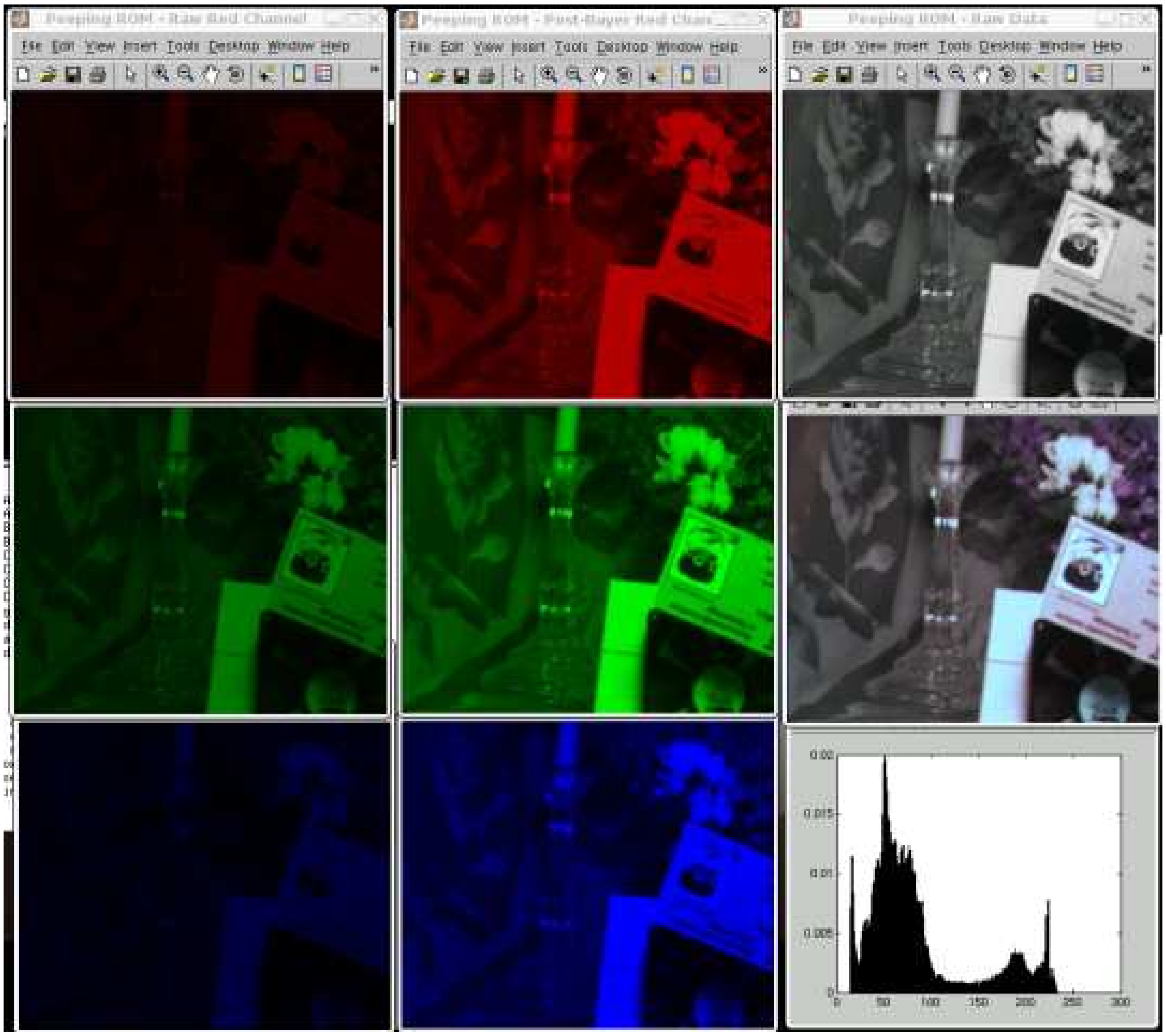
Finally, once all of the pixels have been generated in the red, green, and blue matrices each of these matrices are combined into a single $349 \times 288 \times 3$ matrix that is the full-color image data. (`pr_showpics.m` takes each of these channels in the final output matrix and displays them independently, allowing the user to see the effects of interpolation.)

As a reference, the code listing for `bayer.m` is shown in article 23 of appendix B. Please consult this for more information on the actual interpolation technique.

3.5 Using Peeping ROM's Visual Output

Since Peeping ROM is designed to be an educational tool, it provides visual output at every stage of the image capture process. The following table describes each of the nine output windows that are opened (via `pr_showpics`) after an image is downloaded. The names in brackets in the third column are the actual variable names used in `picture_me` and `automode` to store the image data. For information on accessing these variables after program execution and picture capture, please see section 3.6.

Pre-Interpolation Pixels, Blue Channel	Post-Interpolation Pixels, Blue Channel	Raw Data Pixels <`raw_pic`>
Pre-Interpolation Pixels, Green Channel	Post-Interpolation Pixels, Green Channel	Final Color Image <`color_pic`>
Pre-Interpolation Pixels, Red Channel	Post-Interpolation Pixels, Red Channel	Brightness Histogram <`hist`>


Peeping ROM - Raw Red Channel
Peeping ROM - Post-Bayer Red Chan...
Peeping ROM - Raw Data
File Edit View Insert Tools Desktop Window Help
0.02
0.015
0.01
0.005
0
0 50 100 150 200 250 300

Peeping ROM visual output

Pre-Interpolation Images

`pr_showpics`, the default image display function for Peeping ROM, presents the user with four pre-interpolation images – the raw pixel image and each of the three color channels. One should note that all of these images show particular patterns and graininess that is caused by the presence of the Bayer color filters. Also noteworthy is the fact that all of the single-color channels are significantly darker than their post-interpolation counterparts. This is caused by the fact that in any arbitrary 2×2 group of pixels, one quarter is blue, a second quarter red, and half green. This can be seen visually by using zooming in on the pre-interpolation images until the individual pixel patterns are visible. At this level, the Bayer patterning will be plainly evident – only the pixels of a particular channel's color will be visible. All other pixels will be black since there is no data for the selected channel available.

Post-Interpolation Images

The four post-interpolation pictures are interesting because they show the effects of Peeping ROM's simple interpolation algorithm. The single color figures are interesting because they no longer show any pixelation caused by a lack of image data. However, blurring and image artifacts can appear as a result of the basic interpolation scheme.

Brightness Histogram

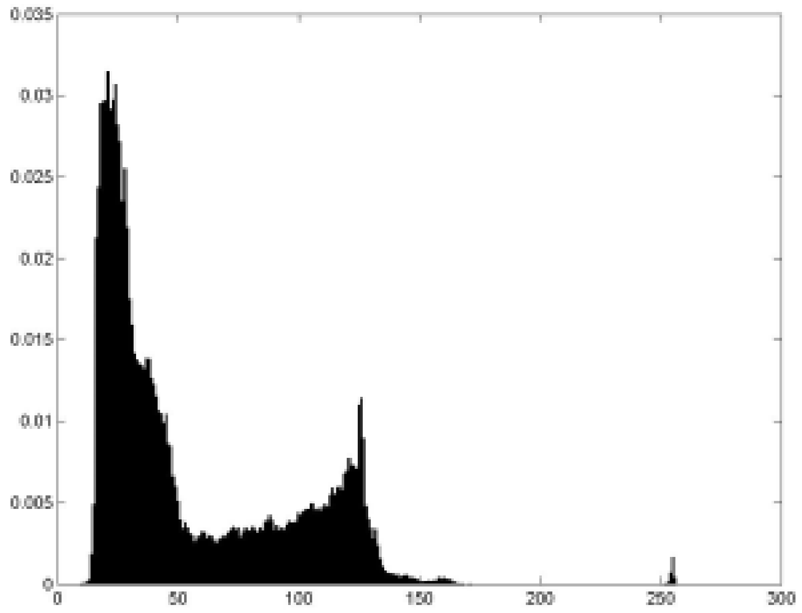
On many modern digital cameras, a brightness histogram can be displayed to show information about the estimated exposure level of a photograph. Our technique, while not being identical to those in true cameras, shows the distribution of pixel intensity values in the final full-color image. Since this image is represented in Matlab as a matrix of 349×288 pixels by 3 channels, we perform the following conversion to generate a matrix that has the effective brightness of each pixel. This algorithm takes into account the human eye's differing sensitivity to red, green, and blue wavelengths.

$$P_{brightness} = (.299 \cdot P_{red}) + (.587 \cdot P_{green}) + (.114 \cdot P_{blue})$$

This algorithm is run on each pixel in the final image, giving a single 349×288 matrix containing the brightness of each pixel. These values, after being rounded to the nearest integer, contain values from 0 (pure black) to 255 (pure white). The histogram function then compares each of these values with every number from 0 to 255 and records how many of each value it finds. This data is then normalized (divided by the total number of pixels in the image, 100,512) and diplayed as a bar graph.

Why do this? As noted earlier, a brightness histogram provides useful information on the overall exposure of the photograph. If the peak of the histogram is located close to the left, it indicates that there are many more dark pixels than light ones and the image is probably underexposed. The converse is also true, with a peak around 255 meaning that the image is blooming. A secondary benefit of the histogram is that it gives insight into the values contained in the picture data matrices without having to actually go through them. While `histogram.m` itself is a specialized function made specifically for the 3 channel color image matrix, the brightness sorting routine can be adapted in other user-written functions to look at pixel value frequencies in other matrixes.

For more information and code listings of `histogram.m`, please see article 24 of appendix B.



Histogram for Underexposed Image

3.6 Accessing Image Data Matrices

Peeping ROM is designed to display a number of images and a composite brightness histogram. However, there may be times when a more in-depth analysis of image data is desired. For this reason Peeping ROM makes three of its final data matrices accessible to the user: `raw_pic`, `color_pic`, and `hist`. These matrices are provided in that order as output values from the `picture_me` function. The user can make these available by calling `picture_me` with the following syntax:

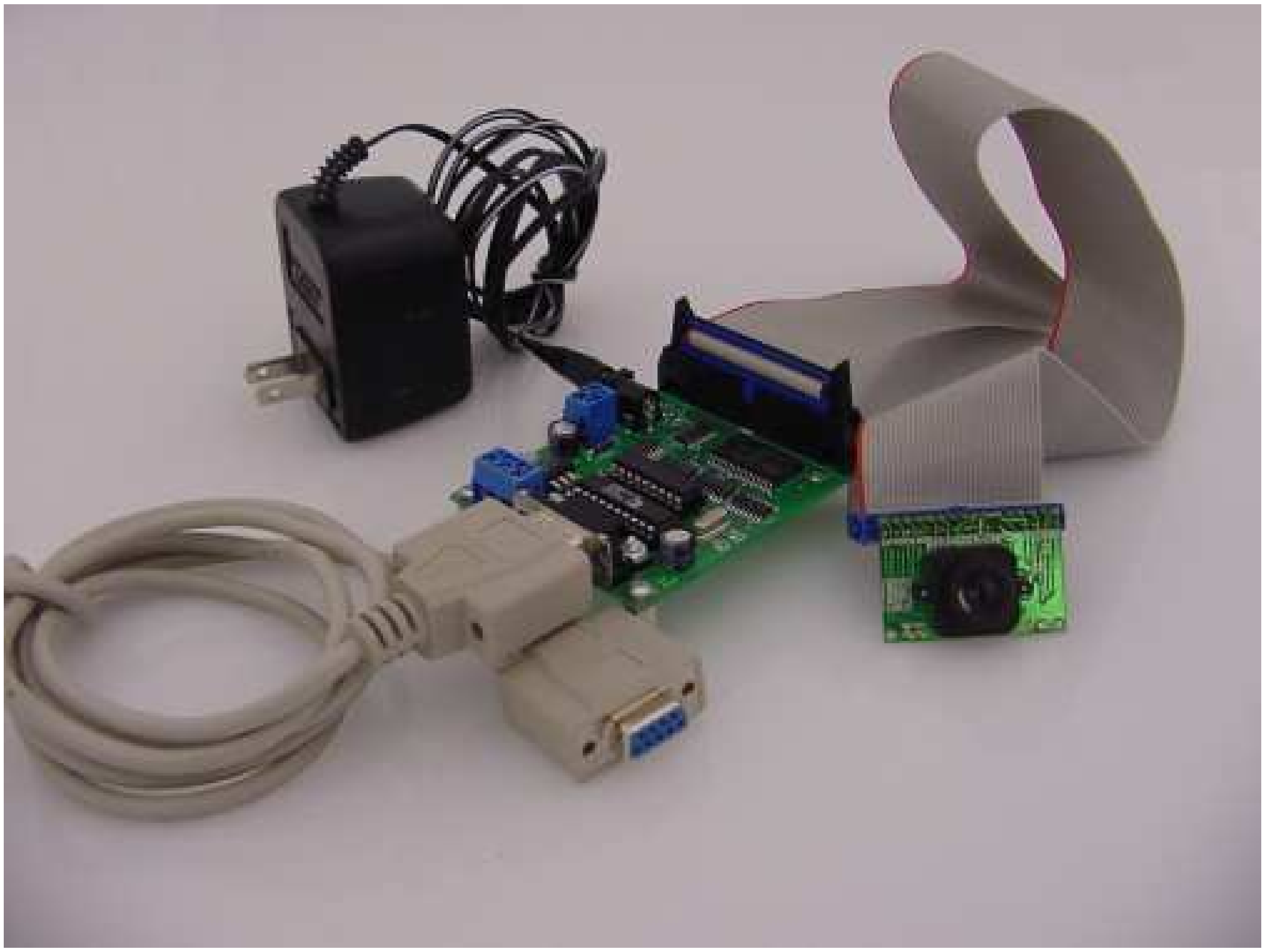
```
[variable_1 variable_2 variable_3] = picture_me;
```

variable_1 will contain `raw_pic`, a 349 × 288 × 1 matrix containing the actual data (0-255) read off the camera at each pixel location. *variable_2* will contain `color_pic`, the post-interpolation 349 × 288 × 3 matrix containing the value for each pixel (0-255) in the final image. Finally, *variable_3* will contain `hist`, a 256 × 1 matrix containing the raw brightness histogram data. Its values will be floating point numbers between 0 and 1. Alternatively, the user can ignore the output from `picture_me` by leaving out any of the material before the equals sign in the command listing shown above.

A Appendix A: The Hardware Components

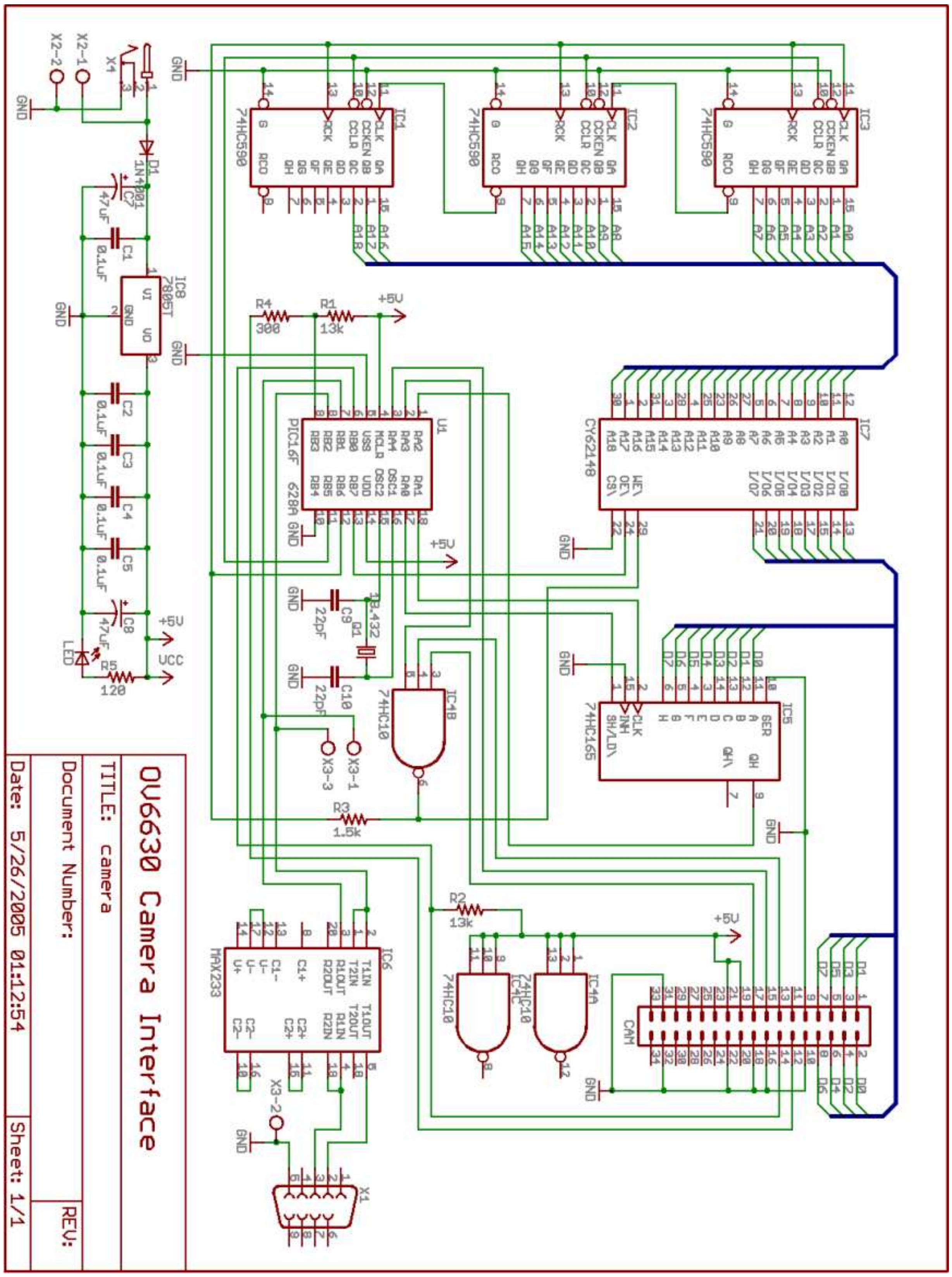
The Peeping ROM hardware acts as an interface between the user (operating with high-level commands in Matlab) and the camera module itself. It features:

- Omnivision OV6630 Color CMOS camera module (349 $\times$ 288 resolution)
- 4 Mbit SRAM for image capture
- TTL and RS-232 serial inputs
- PIC16F628A host microcontroller
- +8-35V input requirement
- 18.432 MHz clock



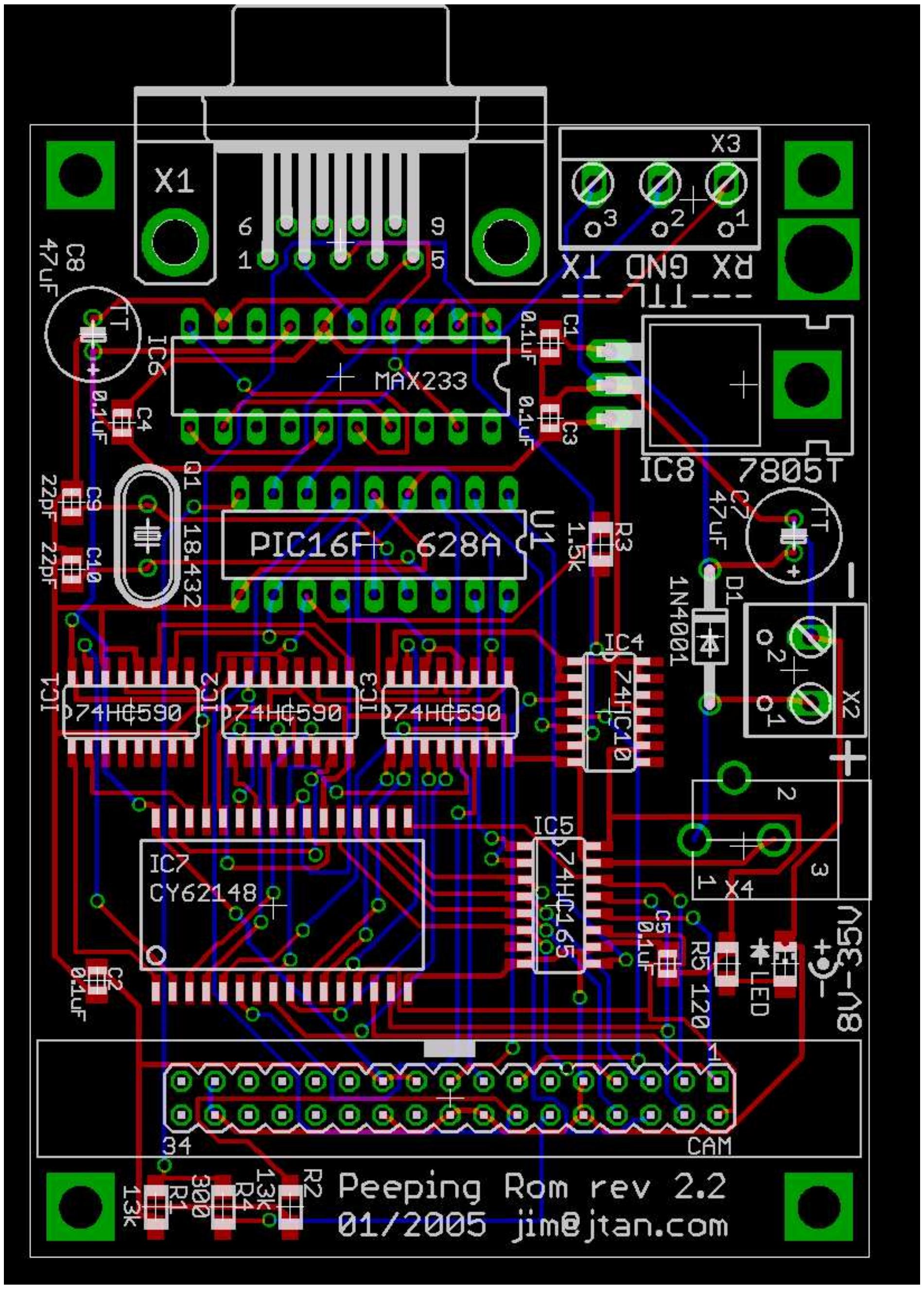
The Peeping ROM System

A.1 Schematic



Peeping ROM schematic

A.2 Board Layout



Peeping ROM board layout

A.3 Potential Flash Trigger Signals

Adding a flash trigger may be done using pin 2 (`RA3`) on the PIC controller board. This TTL signal is driven by the PIC to enable writes into the onboard RAM of Peeping ROM. However, its usability as a flash trigger may be limited by the long exposure times of the camera module since Peeping ROM is not capable of generating a full frame of data faster than 45 frames/second.

A.4 Parts List

Every component necessary for building the Peeping ROM hardware component, with part numbers at Digikey.

part number	**Digikey number**	**Part Description**
IC1	296-8340-5-ND	74HC590 counter
IC2	296-8340-5-ND	74HC590 counter
IC3	296-8340-5-ND	74HC590 counter
IC4	296-4360-1-ND	74AC10 3-input NAND
IC5	296-12791-5-ND	74HC165 shift register
IC6	MAX233CPP-ND	MAX233 RS232 transceiver
IC6-S	AE7220-ND	20 pin socket
IC7	428-1075-ND	CY62148 4Mbit SRAM
IC8	LM7805CT-ND	5V positive regulator
U1	PIC16F628A-I/P-ND	PIC16F628A
U1-S	AE7218-ND	18 pin socket
Q1	X179-ND	18.432 MHz crystal
X1	A2100-ND	Female DB9 socket
X2	ED1601-ND	Terminal block
X3	ED1602-ND	Terminal block
X4	CP-102A-ND	DC power jack 2.1mm
CABLE	C3DDG-3418G-ND	IDC 34-pin, 18 in
CAM	AHL34G-ND	34-pin shrouded header
R1	311-13KECT-ND	13kΩ resistor 1206
R2	311-13KECT-ND	13kΩ resistor 1206
R3	311-1.5KECT-ND	1.5kΩ resistor 1206
R4	311-300ECT-ND	300Ω resistor 1206
R5	311-120ECT-ND	120Ω resistor 1206
D1	1N4001GICT-ND	1N4001 diode
LED	67-1357-1-ND	Green LED 1206
C1	BC1300CT-ND	.1μF ceramic cap 0805
C2	BC1300CT-ND	.1μF ceramic cap 0805
C3	BC1300CT-ND	.1μF ceramic cap 0805
C4	BC1300CT-ND	.1μF ceramic cap 0805
C5	BC1300CT-ND	.1μF ceramic cap 0805
C7	P836-ND	47μF electrolytic cap
C8	P836-ND	47μF electrolytic cap
C9	BC1260CT-ND	22pF ceramic cap 0805
C10	BC1260CT-ND	22pF ceramic cap 0805

A.5 PIC Microcontroller Code

The Peeping ROM system uses a standard PIC16F628A running on an external 18.432 MHz crystal to act as the interface between the camera module, memory, and computer. All of the Matlab functions discussed in section 4 and appendix B send data to this chip to be interpreted and turned into the proper SCCB protocol that the camera then uses.

As a side note, there should be no need for the user to change any of this code. It is provided solely as a reference.

A.5.1 cam.c

`cam.c` provides several of the necessary support routines that allow for register access over the SCCB bus, initialize the camera to safe starting values, and reset the device. It uses `cam.h`, `sccb.h`, and `reg.h`.

```
#ifdef PIC
#include <pic.h>
#endif
#include "sccb.h"
#include "cam.h"

#include "reg.h"

uint8 cam_get(uint8 reg)
{
        return sccb_get(0xC0, reg);
}

void cam_set(uint8 reg, uint8 val)
{
        sccb_put(0xC0, reg, val);
}

void cam_set_safe(uint8 reg, uint8 val)
{
        /* Like cam_set, but makes sure particular values are set
           the correct way to avoid hardware damage.
           Returns actual value written, or -1 if it was disallowed. */
        unsigned char i;

        if(reg > 0x5c) return;

        for(i=0; disallowed_6630[i] < reg; i++)
                continue;
        if(disallowed_6630[i] == reg)
                return;

        for(i=0; safe_6630[i].reg < reg; i++)
                continue;
        if(safe_6630[i].reg == reg) {
                val &= safe_6630[i].mask;
                val |= safe_6630[i].set;
        }

        cam_set(reg, val);
}
```

```
/* Detect, reset, and initialize the chip. Only returns on success */
void cam_reset(void)
{
        uint8 i, j;

        sccb_init();

        for(;;) {
                /* Reset chip */
                cam_set(COMA, 0x80);

                /* Delay 1 second. Camera needs time to settle.. */
                for(i=0; i<100; i++)
                        for(j=0; j<100; j++)
                                sccb_delay(100);

                /* Check ID */
                if(cam_get(MIDH) == 0x7F &&
                   cam_get(MIDL) == 0xA2)
                        break;
        }
        for(i=0; init_6630[i].reg != 0xff; i++)
                cam_set(init_6630[i].reg, init_6630[i].val);
}

/* Set the Y/UV tristate bit, and return nonzero if that bit is now set */
uint8 cam_tristate(uint8 hiz)
{
        uint8 v;

        v = cam_get(COMB);
        if(hiz)
                v |= 0x04;
        else
                v &=~0x04;
        cam_set(COMB, v);
        return cam_get(COMB) & 0x04;
}
```

A.5.2 sccb.c

`sccb.c` is the actual SCCB (Serial Camera Control Bus) implementation on the PIC. It contains 7 functions that initialize the bus, set start/stop conditions, and allow bidirectional communication between the camera and the PIC. It uses `sccb.h` and `serial.h`. For more information on the SCCB bus protocol please see article 2 of appendix C.

```
/* Talk to the camera's SCCB bus, which is basically a subset of I2C. */

#ifdef PIC
#include <pic.h>
#endif
#include "sccb.h"
#include "serial.h"

/* Timings, in us */
#define T_SCL_HIGH 10 /* min clock high period */
#define T_SCL_LOW 10 /* min clock low period */
#define T_START_SETUP 25 /* setup before start */

#if 0
#define T_BUS_FREE 5*5 /* bus free between transfers */
#define T_START_HOLD 5*4 /* hold time for start condition */
#define T_STOP_SETUP 5*4 /* setup before stop */
#define T_SCL_LOW 5*5 /* min clock low period */
#define T_SCL_HIGH 5*4 /* min clock high period */
#define T_SCL_TO_DATA 5*4 /* SCL low to data valid, for reads */
#define T_DATA_SETUP 5*1 /* data setup */
#endif

/*
 * Internal low-level functions
 */

/* Send start condition */
void sccb_start(void)
{
        /* Ensure that we're floating clock high, data high */
        SCL = 0;
        SCL_TRI = 1;
        SDA = 1;
        SDA_TRI = 0;
        sccb_delay(T_START_SETUP);

        /* Float clock high, then pull data low */
        SCL_TRI = 1;
        sccb_delay(T_SCL_HIGH/2);
        SDA = 0;   /* high -> low with SCL = 1 */
        sccb_delay(T_SCL_HIGH/2);
}

/* Send stop condition */
void sccb_stop(void)
{
        /* Ensure that we're pulling data low, clock low */
        SCL = 0;
        SCL_TRI = 0;
        sccb_delay(T_SCL_LOW/2);
        SDA = 0;
        SDA_TRI = 0;
        sccb_delay(T_SCL_LOW/2);

        /* Float clock high, then pull data high */
        SCL_TRI = 1;
        sccb_delay(T_SCL_HIGH/2);
        SDA = 1;   /* low -> high with SCL = 1 */
        sccb_delay(T_SCL_HIGH/2);

        /* Float data */
        SDA_TRI = 1;
}

/* Send raw byte to slave. Returns acknowledgement bit */
uint8 sccb_send(uint8 data)
{
        uint8 i;

        for(i=8; i>0; i--) {
                /* Drive clock low, then set data */
                SCL = 0;
                SCL_TRI = 0;

                sccb_delay(T_SCL_LOW/2);

                SDA = (data & 0x80) ? 1 : 0;
                SDA_TRI = 0;
                data <<= 1;

                sccb_delay(T_SCL_LOW/2);

                /* Float clock high */
                SCL_TRI = 1;

                /* Wait for the slave to let it go */
                while(SCL == 0) continue;

                /* Now make sure it stays high for a bit */
                sccb_delay(T_SCL_HIGH);
        }

        /* Now read acknowledgement back. Set clock low and tristate data */
        SCL = 0;
        SCL_TRI = 0;
        SDA_TRI = 1;

        sccb_delay(T_SCL_LOW);

        /* Float clock high, wait for slave to let it go */
        SCL_TRI = 1;
        while(SCL == 0) continue;

        /* Read the bit and return it */
        sccb_delay(T_SCL_HIGH/2);
        i = SDA;
        sccb_delay(T_SCL_HIGH/2);
        return i;
}

/* Read raw byte from slave. Replies with acknowledgement bit. */
uint8 sccb_recv(uint8 ack)
{
        uint8 i, data = 0;

        for(i=8; i>0; i--) {
                /* Drive clock low, then release data */
                SCL = 0;
                SCL_TRI = 0;
                SDA_TRI = 1;
                sccb_delay(T_SCL_LOW);

                /* Float clock high, wait for slave */
                SCL_TRI = 1;
                while(SCL == 0) continue;

                /* Wait and read the byte */
                sccb_delay(T_SCL_HIGH);
                data <<= 1;
                data |= SDA;
        }

        /* Send acknowledgement. Drive clock low and set data. */
        SCL = 0;
        SCL_TRI = 0;
        sccb_delay(T_SCL_LOW/2);
        SDA = (ack & 1);
        SDA_TRI = 0;
        sccb_delay(T_SCL_LOW/2);

        /* Float clock high, then we're done */
        SCL_TRI = 1;
        sccb_delay(T_SCL_HIGH);

        return data;
}

/*
 * Public high-level interface
 */

/* Initialize SCCB bus */
void sccb_init(void)
{
        SCL_TRI = 1;
        SDA_TRI = 1;
}

/* Put a byte to register "subaddress" in chip "address" with data "data" */
/* Acknowledgement bits are ignored. */
void sccb_put(uint8 address, uint8 subaddress, uint8 data)
{
        sccb_start();
        sccb_send(address | 0x00);
        sccb_send(subaddress);
        sccb_send(data);
        sccb_stop();
}

/* Get a byte from register "subaddress" in chip "address" */
uint8 sccb_get(uint8 address, uint8 subaddress)
{
        uint8 data = 0;

        sccb_start();
        sccb_send(address | 0x00);
        sccb_send(subaddress);
        sccb_stop();
        sccb_start();
        sccb_send(address | 0x01);
        data = sccb_recv(SCCB_NAK);
        sccb_stop();
        return data;
}
```

A.5.3 serial.c

`serial.c` provides support for the PIC's onboard UART. This allows the PIC to communicate with Matlab on the host PC through a serial connection. It uses `serial.h`.

```
#ifdef PIC
#include <pic.h>
#endif
#include "serial.h"

/* Initialize serial port */
void serial_init(void)
{
        int i;
        TRISB1 = 1;
        TRISB2 = 0;

        BRGH = 1;
#define BRG (((FOSC + (8 * BAUDRATE − 1)) /(16 * BAUDRATE)) − 1)
#if (BRG < 1) || (BRG > 255)
#error Cannot achieve baudrate
#endif
#define ACT_BR (FOSC / (16 * (BRG + 1)))
#if ((ACT_BR * 100 / BAUDRATE) > 105) || ((ACT_BR * 100 / BAUDRATE) < 94)
#error Actual baudrate is too far from requested baudrate. Ratio is
#error FOSC/(16*floor((FOSC+(8*BAUDRATE−1))/(16*BAUDRATE)))/BAUDRATE
#endif

        SPBRG = BRG;
        SYNC = 0;
        SPEN = 1;
        CREN = 1;
        SREN = 0;
        TXIE = 0;
        RCIE = 0;
        TX9 = 0;
        RX9 = 0;
        TXEN = 1;
        for(i=0; i<100; i++)
                continue;
}

/* Send byte, blocking */
void serial_put(unsigned char x)
{
        while(!TXIF)
                ;
        TXREG = x;
}

/* Send string, blocking */
void serial_put_string(const uint8 *s)
{
        int i;
        for(i=0; s[i]; i++)
                serial_put(s[i]);
}
```

```
/* Send CRLF */
void serial_crlf(void)
{
        serial_put_2('\r','\n');
}

const uint8 hex[16]={'0','1','2','3','4','5','6','7',
                     '8','9','a','b','c','d','e','f'};

void serial_put_hex(unsigned char x)
{
        serial_put(hex[x >> 4]);
        serial_put(hex[x & 15]);
}

/* Get byte, blocking */
unsigned char serial_get(void)
{
        for(;;) {
                if(RCIF) {
                        return RCREG;
                }
                if(OERR) {
                        CREN = 0;
                        CREN = 1;
                }
        }
}

unsigned char from_hex_digit(unsigned char c)
{
        if(c < '0')
                goto bad;
        if(c <= '9')
                return c - '0';
        c |= 0x20;
        if(c < 'a')
                goto bad;
        if(c <= 'f')
                return c - 'a' + 10;
 bad:
        return 16;
}

/* Get hex byte, return value is byte or -1 for invalid input */
signed int serial_get_hex(void)
{
        unsigned char h, l;

        h = from_hex_digit(serial_get());
        if(h & 16)
                return -1;
        l = from_hex_digit(serial_get());
        if(l & 16)
                return -1;
        h = (h << 4) | (h >> 4);
        return h|l;
}
```

A.5.4 main.c

`main.c` contains the program that is running continually while Peeping ROM is powered up. It initializes the camera and serial port, waits for characters that specify what action to take, and commands the camera to change its register settings or snap a picture into memory. It also reads the memory on Peeping ROM to slowly download the picture data to a PC.

```
#ifdef PIC
#include <pic.h>
#endif
#include "config.h"
#include "serial.h"
#include "cam.h"

/* 2 hex digits */
#define VERSION "02"

#define discard_newline serial_get

__CONFIG(CONFIGWORD);

bit downloading = 0;

void main(void) {
        uint8 i, ch, data;
        sint16 r;
        uint8 ah, am, al;

        GIE = 0;
        CMCON = 7;
        TRISA = 0xff;
        TRISB = 0xff;

        VSYNC_TRI = 1;
        WRITE = 0;
        WRITE_TRI = 0;
        ADDR_RESET = 1;
        ADDR_RESET_TRI = 0;
        ADDR_INC_TRI = 1;
        SR_LOAD = 1;
        SR_LOAD_TRI = 0;
        SR_CLOCK = 0;
        SR_CLOCK_TRI = 0;
        SR_DATA_TRI = 1;
        RAM_OE = 1;
        RAM_OE_TRI = 0;

        /* Serial */
        serial_init();
        cam_reset();

        goto version;

        /* Using for(;;) here gives an unreachable code warning,
           so just loop with a goto */
 looptop:
        ch = serial_get();
        ch &= ~0x20; // force uppercase

        /* If we're downloading, the only valid command is 'M'
           (return more data). If we get anything else, stop downloading */
        if(downloading && ch != 'M') {
                ADDR_INC_TRI = 1;
                cam_tristate(0);
                downloading = 0;
        }

        switch(ch) {
        case 'V': /* version */
                discard_newline();
        version:
                serial_put_6('v','e','r',' ',VERSION[0],VERSION[1]);
                break;
        case 'R': /* reinit */
                discard_newline();
                cam_reset();
                serial_put_2('o','k');
                break;
        case 'P': /* take picture */
                discard_newline();
                /* Wait for frame to start */
                while(VSYNC==0)
                        continue;
                /* Reset address and enable counting/writing */
                ADDR_RESET = 0;
                ADDR_RESET = 1;
                ADDR_INC_TRI = 0;
                ADDR_INC = 0;
                ADDR_INC = 1;
                ADDR_INC_TRI = 1;
                RAM_OE = 1;
                WRITE = 1;

                while(VSYNC==1)
                        continue;
                /* frame writing now */
                while(VSYNC==0)
                        continue;

                WRITE = 0;

                /* Make sure we don't catch beginning of next frame */
                while(VSYNC==1)
                        continue;

                serial_put_2('o','k');
                break;

        case 'G': /* get register */

                /* Read register number */
                r = serial_get_hex();
                if(r >= 0) {
                        discard_newline();
                        i = r & 255;
                get_reg:
                        data = cam_get(i);
                } else {
                put_zero:
                        data = 0;
                }
                /* Display value */
                serial_put_hex(data);

                break;

        case 'S': /* set register */

                /* Read register number */
                r = serial_get_hex();
                if(r < 0) goto put_zero;
                i = r & 255;

                /* Read value */
                r = serial_get_hex();
                if(r < 0) goto put_zero;
                discard_newline();
                cam_set_safe(i, r & 255);

                goto get_reg;
                // break;

        case 'D': /* initiate download */

                /* Read starting address (3 hex words, MSB first) */
                r = serial_get_hex();
                if(r < 0) break;
                ah = r;

                r = serial_get_hex();
                if(r < 0) break;
                am = r;

                r = serial_get_hex();
                if(r < 0) break;
                al = r;

                discard_newline();

                /* Tristate camera; hang if it fails */
                if(cam_tristate(1) == 0) {
                        for(;;) continue;
                }

                /* Ensure we're not writing, then reset address */
                WRITE = 0;
                ADDR_RESET = 0;
                ADDR_RESET = 1;
                ADDR_INC_TRI = 0;
                ADDR_INC = 0;
                ADDR_INC = 1;
                SR_LOAD = 1;
                SR_CLOCK = 0;

                /* Advance to the requested address. Decrement is
                 * optimized for speed. */
                while(al != 0) {
                inside:
                        al--;
                        ADDR_INC = 0; ADDR_INC = 1;
                }
                if(am != 0) {
                insidem:
                        am--;
                        goto inside;
                }
                if(ah != 0) {
                        ah--;
                        goto insidem;
                }

                serial_put_2('o','k');
                downloading = 1;
                break;

        case 'M':
                if(!downloading) break;
                discard_newline();

                /* Read out 128 bytes */
                ch=128;
                while(ch-- != 0) {
                        /* Load byte into shift register */
                        RAM_OE = 0;
                        SR_LOAD = 0;
                        SR_LOAD = 1;
                        RAM_OE = 1;
                        /* Clock it in and send via serial */
                        data = 0;
                        for(i=0; i<8; i++) {
                                data <<= 1;
                                data |= SR_DATA;
                                SR_CLOCK = 1;
                                SR_CLOCK = 0;
                        }
                        serial_put_hex(data);
                        if(ch)
                                serial_put(' ');

                        /* Increment address */
                        ADDR_INC = 0;
                        ADDR_INC = 1;
                }
                break;

        default:
                break;
        }
```

```
        serial_put('\n');

        goto looptop;
}
```

A.5.5 Header Files: cam.h, sccb.h, serial.h, config.h, reg.h

cam.h: provides structures and prototypes for `cam.c`

```
#ifndef CAM_H
#define CAM_H

#include "config.h"

struct cam_regvals {
        unsigned char reg;
        unsigned char val;
};

struct cam_safevals {
        unsigned char reg;
        unsigned char mask;
        unsigned char set;
};

typedef enum {
        GAIN = 0x00, BLUE, RED, SAT,
        CTR = 0x05, BRT, SHP,
        ABLU = 0x0C, ARED, COMR, COMS, AEC, CLKRC, COMA, COMB, COMC, COMD,
        FSD = 0x16, HREFST, HREFEND, VSTRT, VEND, PSHFT, MIDH, MIDL,
        COME = 0x20, YOFF, UOFF, CLKC, AEW, AEB, COMF, COMG, COMH, COMI,
        FRARH = 0x2A, FRARL,
        COMJ = 0x2D, VCOFF,
        CPP = 0x33, BIAS,
        COMK = 0x38, COML, HSST, HSEND, COMM, COMN, COMO, COMP,
        YMXA = 0x4D, ARL, YMXB,
        COMQ = 0x54,
        DBL = 0x57,
        OFC = 0x59, SC, SAWB } cam_reg;

void cam_reset(void);
uint8 cam_get(uint8 reg);
void cam_set(uint8 reg, uint8 val);
void cam_set_safe(uint8 reg, uint8 val);
uint8 cam_tristate(uint8 hiz);

#endif
```

sccb.h: provides function prototypes and provides delay functions that vary depending on oscillator speed variable `FOSC`.

```
/* Talk to the camera's SCCB bus, which is basically a subset of I2C. */

#ifndef SCCB_H
#define SCCB_H

#include "config.h"

#if !(defined SDA && defined SCL && defined SDA_TRI && defined SCL_TRI)
#error Define SDA, SCL, SDA_TRI, and SCA_TRI in config.h
#endif


#if FOSC >= 12000000
#define sccb_delay(x) do { uint8 _d = (x) * (FOSC / 12000000); \
                          while(--_d != 0) continue;} while(0)
#else
#define sccb_delay(x) do { uint8 _d = (x) / (12000000 / FOSC) | 1; \
                          while(--_d != 0) continue;} while(0)
#endif

#define SCCB_NAK 1
#define SCCB_ACK 0

/* Internal functions */
void sccb_start(void);
void sccb_stop(void);
uint8 sccb_send(uint8 data);
uint8 sccb_recv(uint8 ack);

/* Public */

/* Initialize SCCB bus */
void sccb_init(void);
/* Put a byte to register "subaddress" in chip "address" with data "data" */
void sccb_put(uint8 address, uint8 subaddress, uint8 data);
/* Get a byte from register "subaddress" in chip "address" */
uint8 sccb_get(uint8 address, uint8 subaddress);

#endif
```

serial.h: provides function prototypes and serial communication hacks to get around limited programming space availability from the free compiler.

```
#ifndef SERIAL_H
#define SERIAL_H

#include "config.h"

/* Initialize serial port */
void serial_init(void);

/* These are smaller than the equivalent serial_put_string call */
#define serial_put_1(a) serial_put(a)
#define serial_put_2(a,b) serial_put_1(a),serial_put_1(b)
#define serial_put_3(a,b,c) serial_put_1(a),serial_put_2(b,c)
#define serial_put_4(a,b,c,d) serial_put_1(a),serial_put_3(b,c,d)
#define serial_put_5(a,b,c,d,e) serial_put_1(a),serial_put_4(b,c,d,e)
#define serial_put_6(a,b,c,d,e,f) serial_put_1(a),serial_put_5(b,c,d,e,f)

/* Send byte, blocking */
void serial_put(unsigned char x);
void serial_put_string(const uint8 *s);
void serial_crlf(void);
void serial_put_hex(unsigned char x);

/* Get byte, blocking */
unsigned char serial_get(void);
signed int serial_get_hex(void);

#define serial_can_get() (RCIF)

#endif
```

config.h: provides constant definitions, pin aliases, and the PIC config word.

```
#ifndef CONFIG_H
#define CONFIG_H

#define CONFIGWORD (WDTDIS & PWRTEN & MCLREN & BOREN & LVPEN & HS)

#define FOSC 18432000
/* Max standard baudrate with FOSC=4000000 is 19200 */
/* Max standard baudrate with FOSC=18432000 is 230400 */
/* Max standard baudrate with FOSC=20000000 is 115200 */
#define BAUDRATE 115200L

#define SDA RB3
#define SDA_TRI TRISB3
#define SCL RB0
#define SCL_TRI TRISB0
#define ADDR_INC RB6
#define ADDR_INC_TRI TRISB6
#define ADDR_RESET RB5
#define ADDR_RESET_TRI TRISB5
#define WRITE RA3
#define WRITE_TRI TRISA3
#define VSYNC RA4
#define VSYNC_TRI TRISA4
#define SR_LOAD RA0
#define SR_LOAD_TRI TRISA0
#define SR_CLOCK RA1
#define SR_CLOCK_TRI TRISA1
#define SR_DATA RA2
#define SR_DATA_TRI TRISA2
#define RAM_OE RB7
#define RAM_OE_TRI TRISB7

typedef unsigned char uint8;
typedef signed char sint8;
typedef unsigned int uint16;
typedef signed int sint16;
typedef unsigned long uint32;
typedef signed long sint32;

#endif
```

reg.h: provides the initialization value for every register on the OV6630 camera module, as well as masks to prevent user changes to certain registers that may damage the camera physically. These settings come from the Linux ov511 driver with modifications for better picture quality by Jim Paris.

Note: the initialization values shown below may not be the same values that would be read out of the camera immediately after initialization as some registers act indirectly on others. Please consult pages 20-27 of the OV6630 datasheet in article 1 of appendix C for more information about the registers and their actions.

```
/* Camera register data, used by cam.c */

/* Safe register settings.
   If register matches first value:
      Second value is mask to apply (~0xFF means turn off all bits)
      Third value is bits to force on
*/
const static struct cam_safevals safe_6630[] = {
/* Allow changes to CLKRC[5:0]; if it's set to zero,
   the rate will likely be too fast for the RAM, but
   it shouldn't break. */
        { 0x11, ~0xc0, 0x00 }, /* Force pixel clock settings */
        { 0x12, ~0x80, 0x08 }, /* Disallow reset, force RGB */
        { 0x13, ~0xfc, 0x00 }, /* Force output format and tristate */
        { 0x20, ~0x81, 0x00 }, /* Don't change HREF stuff, driver strength */
        { 0x27, ~0xf0, 0xa0 }, /* Force CKOUT settings */
        { 0x28, ~0xc4, 0x80 }, /* keep RGB format correct */
        { 0x29, ~0x40, 0x00 }, /* master mode */
        { 0x38, ~0xf0, 0x80 }, /* href settings, no overdrive */
        { 0xff, 0xff, 0xff }, /* end marker */
};

/* Registers to which changes are just plain disallowed */
const static unsigned char disallowed_6630[] = {
        0x14, 0x15, 0x16, /* Don't mess with output format */
        0x23, /* oscillator control */
        0x39, 0x3e, /* more href and timing stuff */
        0x3f, /* clocks and timing */
        0x57, /* charge pump */
        0xff /* end marker */
};

/* Initialization. This sequence is always sent to the camera after a
   reset. Taken from the Linux ov511 driver, with modifications
   marked with "jim" */
const static struct cam_regvals init_6630[] = {
        { 0x12, 0x80 }, /* Reset */
        { 0x00, 0x1f }, /* Gain */
        { 0x01, 0x99 }, /* Blue gain */
        { 0x02, 0x7c }, /* Red gain */
        { 0x03, 0xc0 }, /* Saturation */
        { 0x05, 0x0a }, /* Contrast */
        { 0x06, 0x95 }, /* Brightness */
        { 0x07, 0x2d }, /* Sharpness */
        { 0x0c, 0x20 },
        { 0x0d, 0x20 },
        { 0x0e, 0x20 },
{ 0x0f, 0x05 },
{ 0x10, 0x9a }, /* "exposure check" */
{ 0x11, 0x01 }, /* Pixel clock = fastest */ /* jim: div by 2 */
{ 0x12, 0x2C }, /* Enable AGC and AWB */ /* jim: select RGB */
{ 0x13, 0x01 }, /* jim: 16-bit format (8 bit is 0x21) */
{ 0x14, 0x80 },
{ 0x15, 0x01 },
{ 0x16, 0x03 },
{ 0x17, 0x38 },
{ 0x18, 0xe8 }, /* jim: reduce to e8 to keep horiz width 352 */
{ 0x19, 0x04 },
{ 0x1a, 0x93 },
{ 0x1b, 0x00 },
{ 0x1e, 0xc4 },
{ 0x1f, 0x04 },
{ 0x20, 0x20 },
{ 0x21, 0x00 }, /* jim: don't change y offset */
{ 0x22, 0x00 }, /* jim: don't change u offset */
{ 0x23, 0xc0 }, /* Crystal circuit power level */
{ 0x25, 0x9a }, /* Increase AEC black pixel ratio */
{ 0x26, 0xb2 }, /* BLC enable */
{ 0x27, 0xa2 },
{ 0x28, 0x80 }, /* jim: one-line RGB */
{ 0x29, 0x00 },
{ 0x2a, 0x84 }, /* (keep) */
{ 0x2b, 0xa8 }, /* (keep) */
{ 0x2c, 0xa0 },
{ 0x2d, 0x95 }, /* Enable auto-brightness */
{ 0x2e, 0x00 }, /* jim: don't change v offset */
{ 0x33, 0x26 },
{ 0x34, 0x03 },
{ 0x36, 0x8f },
{ 0x37, 0x80 },
{ 0x38, 0x83 },
{ 0x39, 0x80 },
{ 0x3a, 0x0f },
{ 0x3b, 0x3c },
{ 0x3c, 0x1a },
{ 0x3d, 0x80 },
{ 0x3e, 0x80 },
{ 0x3f, 0x0e },
{ 0x40, 0x00 }, /* White bal */
{ 0x41, 0x00 }, /* White bal */
{ 0x42, 0x80 },
{ 0x43, 0x3f }, /* White bal */
{ 0x44, 0x80 },
{ 0x45, 0x20 },
{ 0x46, 0x20 },
{ 0x47, 0x80 },
{ 0x48, 0x7f },
{ 0x49, 0x00 },
{ 0x4a, 0x00 },
{ 0x4b, 0x80 },
{ 0x4c, 0xd0 },
{ 0x4d, 0x10 }, /* U = 0.563u, V = 0.714v */
{ 0x4e, 0x40 },
{ 0x4f, 0x07 }, /* UV average mode, color killer: strongest */
```

```
        { 0x50, 0xff },
        { 0x54, 0x23 }, /* Max AGC gain: 18dB */
        { 0x55, 0xff },
        { 0x56, 0x12 },
        { 0x57, 0x89 }, /* (default) */ /* jim: no VSYNC in dropped field */
        { 0x58, 0x75 },
        { 0x59, 0x01 }, /* AGC dark current compensation: +1 */
        { 0x5a, 0x2c },
        { 0x5b, 0x0f }, /* AWB chrominance levels */
        { 0x5c, 0x10 },
        { 0x3d, 0x80 },
        { 0x27, 0xa6 },
        /* Toggle AWB off and on */
        { 0x12, 0x28 }, /* jim: RGB mode */
        { 0x12, 0x2C }, /* jim: RGB mode */

        { 0xff, 0xff }, /* END MARKER */
};
```

B Appendix B: The Matlab Components

These Matlab scripts provide an interface that communicates with Peeping ROM's hardware to command the camera. For usage information on these functions, please see "Using the Peeping ROM System" in section 3.

Filename	**Functionality**
`test_me.m`	fully automatic image capture program
`picture_me.m`	image capture program for user modifications
`camera_setup.m`	Peeping ROM initialization wrapper
`camera_usermode.m`	Peeping ROM user modification wrapper
`camera_takepic.m`	Peeping ROM image capture and download wrapper
`pr_bitset.m`	bit setting function
`pr_check.m`	camera detection function
`pr_close.m`	serial port closure function
`pr_display.m`	figure window generation function
`pr_dlchunk.m`	robust data packet downloader
`pr_dlsetup.m`	download setup function
`pr_download.m`	data download wrapper function
`pr_flush.m`	serial buffer flush function
`pr_get.m`	camera register reader function
`pr_init.m`	serial port configuration function
`pr_modeset.m`	automatic mode configuration function
`pr_picture.m`	image capture function
`pr_regset.m`	register manipulation function
`pr_regread.m`	register value display function
`pr_reinit.m`	camera reinitialization function
`pr_set.m`	register writer function
`pr_showpics.m`	image data display function
`bayer.m`	bayer pattern interpolation function
`histogram.m`	brightness histogram generator function

B.1 test_me.m

NAME:	`test_me` – fully-automatic image capture program
CALL:	**test_me**()
INPUTS:	None.
OUTPUTS:	None.
DESCRIPTION:	`test_me` is a program that takes a picture with all of the Peeping ROM's automatic modes enabled. During the 10 second register delay period it provides a verbose output that shows the ideal register settings to use when making modifications later, making it useful for capturing baseline pictures.
EXAMPLES:	Typical usage – capture baseline photo.

```
test_me;
```

SEE ALSO:	sec 3.1

```
function data = test_me()
    %% takes entirely automatic picture
    %% displays all data in verbose mode
    disp('Peeping ROM test function.');
    disp('Hardware by Jim Paris;');
    disp('Software by Jim Paris & Andrew Muth.');
    disp('');

    %% variable setup
    clear raw_pic;
    clear color_pic;
    clear hist;

    %% camera picture algorithm
    camera_setup; % <--- set serial port here, if desired

    %% automatic mode enables
    disp('-----');
    pr_modeset('AEC','enabled','AGC','enabled','AWB','enabled','verbose');
    disp('-----');

    %% display registers during settling period
    disp('Registers displayed:');
    disp('time: 0x01 : 0x02 : 0x05 : 0x06 : 0x07 : 0x10 : 0x00');
    for i = 10 : -1 : 1
        bluegain = pr_get(hex2dec('01'));
        redgain = pr_get(hex2dec('02'));
        contrast = pr_get(hex2dec('05'));
        brightness = pr_get(hex2dec('06'));
        sharpness = pr_get(hex2dec('07'));
        autoexpose = pr_get(hex2dec('10'));
        gain = pr_get(hex2dec('00'));
        disp(sprintf(' %02d: %02X %02X %02X %02X %02X %02X %02X',[i,bluegain,redgain,contrast,
            brightness,sharpness,autoexpose,gain]));
        pause(1);
     end

    %% capture picture
    raw_pic = camera_takepic;

    %% image interpolation algorithm from Jim
    [ color_pic(:,:,1) color_pic(:,:,2) color_pic(:,:,3) red_raw green_raw blue_raw ] = feval('bayer',raw_pic);

    %% display raw pic, color pic, and histogram
    pr_display(raw_pic, 'Raw Data', 1000);
    pr_display(color_pic, 'Final Image', 1001);
    hist = histogram(color_pic, 1002);

    %% done working
    disp('Done.');
```

B.2 picture_me.m

NAME: `picture_me` – image capture program for applying user modifications

CALL: [`out_1 out_2 out_3`] = **picture_me**()

INPUTS: None.

OUTPUTS: `out_1` – M × N × 1 matrix containing raw data from camera.
`out_2` – M × N × 3 matrix containing final full-color image data from camera.
`out_3` – 1 × 256 matrix containing histogram data.

DESCRIPTION: `picture_me` is a user-accessible program designed to take a picture after applying user modifications to camera registers. It contains all of the necessary function calls to set up Peeping ROM, alter registers, capture an image, and display the image data in multiple figures.

EXAMPLES: Typical usage – capture picture and supress output data.

```
picture_me;
```

Typical usage – capture picture and save all output data.

```
[raw_data color_data hist_data] = picture_me;
```

SEE ALSO: sec 3.2

```
function [raw_pic, color_pic, hist] = picture_me()
    %% call picture_me to take a picture
    disp('Peeping ROM image capture function.');
    disp('Hardware by Jim Paris;');
    disp('Software by Jim Paris & Andrew Muth.');
    disp('');

    %% variable inits
    clear raw_pic;
    clear color_pic;
    clear hist;

    %% camera picture algorithm
    camera_setup; % <--- set serial port here, if desired
    camera_usermode('verbose'); % <--- enter user changes in this function!
    raw_pic = camera_takepic;

    %% image interpolation algorithm from Jim
    [ color_pic(:,:,1) color_pic(:,:,2) color_pic(:,:,3) red_raw green_raw blue_raw ] = feval('bayer',raw_pic);

    %% display all 8 images and histogram
    pr_showpics(raw_pic, color_pic, red_raw, green_raw, blue_raw);
    hist = histogram(color_pic, 1008);

    %% done working
    disp('Done.');
```

B.3 camera_setup.m

NAME: `camera_setup` – Peeping ROM initialization function

CALL: **camera_setup**([`arg_1`])

INPUTS: `arg_1` – *optional* serial port argument.

OUTPUTS: None.

DESCRIPTION: `camera_setup` is a wrapper that calls all of the necessary intialization functions used to start Peeping ROM. It will pass an optional "serial port name" argument to `pr_init`. It also checks that there is a valid, operational Peeping ROM unit responding on the opened port. Finally, it will reinitialize the camera to make sure all registers are in their correct startup configurations.

EXAMPLES: Typical usage – initialize Peeping ROM with port autodetection.

```
camera_setup;
```

Typical usage – initialize with specified serial port `'COM3'`.

```
camera_setup('COM3');
```

SEE ALSO: sec 3.2.1

```
function data = camera_setup(serialport)
    %%performs basic setup routines for peeping rom

    %% Initialize camera
    if (nargin == 1)
        pr_init(serialport);
    else
        pr_init;
    end

    pr_reinit;
```

B.4 camera_usermode.m

NAME: `camera_usermode` – register modification function

CALL: **camera_usermode**([`arg_1`])

INPUTS: `arg_1` – *optional* register delay period mode specification.

Argument	**Effect**
`'verbose'`	During delay period, values of 7 registers are displayed in Matlab. Registers displayed are `0x01`, `0x02`, `0x05`, `0x06`, `0x07`, `0x10`, and `0x00`.
`'quiet'`	**default mode;** During delay period no output is displayed in Matlab.
`'no-delay'`	Delay period does not occur and picture is taken immediately. **Note:** using this mode is not recommended unless all automatic modes have been disabled and their respective registers set to appropriate values.

OUTPUTS: None.

DESCRIPTION: `camera_usermode` is a wrapper that allows the user to deactivate Peeping ROM's automatic modes and specify modifications to camera registers. It is designed to be used within the `picture_me` framework provided with Peeping ROM.

EXAMPLES: Typical usage – verbose output during delay period.

```
camera_usermode('verbose');
```

Typical usage – disable register delay period.

```
camera_usermode('no-delay');
```

Typical usage – quiet output during delay period.

`camera_usermode('quiet');` or
`camera_usermode;`

SEE ALSO: sec 3.2.2, pr_regset, pr_bitset, pr_get, pr_modeset

```
function data = camera_usermode(mode)
    %% mode setup
    if (nargin == 0)
        mode = 'quiet'
    end

    %% camera automode setup:
    %% set 'enabled' to 'disabled', 'verbose' to 'quiet' if desired
    disp('-----');
    pr_modeset('AEC','enabled','AGC','enabled','AWB','enabled','verbose');
    disp('-----');

    %% perform user-specified camera changes here
    %% for register changes, use:
    %% pr_regset(<reg#>,<value>,['verbose']);
    %% for bitset changes, use:
    %% pr_bitset(<reg#>,<bit#>,<value>,['verbose']);
    %% for register READS only, use:
    %% pr_regread(<reg#>);
    disp('Applying user-specified changes...');
    %% insert changes here
    disp('-----');

    %% register display during register settling period
    %% enable by calling camera_usermode('verbose')
    switch mode
        case 'verbose'
            disp('Registers displayed:');
            disp('time: 0x01 : 0x02 : 0x05 : 0x06 : 0x07 : 0x10 : 0x00');
            for i = 10 : -1 : 1
                bluegain = pr_get(hex2dec('01'));
                redgain = pr_get(hex2dec('02'));
                contrast = pr_get(hex2dec('05'));
                brightness = pr_get(hex2dec('06'));
                sharpness = pr_get(hex2dec('07'));
                autoexpose = pr_get(hex2dec('10'));
                gain = pr_get(hex2dec('00'));
                disp(sprintf(' %02d: %02X %02X %02X %02X %02X %02X %02X',[i,bluegain,redgain,
                    contrast,brightness,sharpness,autoexpose,gain]));
                pause(1);
            end
        case 'quiet'
            disp('Please wait for register setup to finish (10 sec).')
            pause(10);
            disp('Setup completed.');
            case 'no-delay'
                disp('Warning: No delay period enabled!');
            pause(1);
            disp('Setup completed.');
        otherwise
            warning('Unknown argument to camera_usermode - using verbose mode...');
            disp('Registers displayed:');
            disp('time: 0x01 : 0x02 : 0x05 : 0x06 : 0x07 : 0x10 : 0x00');
            for i = 10 : -1 : 1
                bluegain = pr_get(hex2dec('01'));
                redgain = pr_get(hex2dec('02'));
                contrast = pr_get(hex2dec('05'));
                brightness = pr_get(hex2dec('06'));
                sharpness = pr_get(hex2dec('07'));
                autoexpose = pr_get(hex2dec('10'));
                gain = pr_get(hex2dec('00'));
                disp(sprintf(' %02d: %02X %02X %02X %02X %02X %02X %02X',[i,bluegain,redgain,
                    contrast,brightness,sharpness,autoexpose,gain]));
                pause(1);
            end
    end
```

B.5 camera_takepic.m

NAME:	`camera_takepic` – image capture and download function
CALL:	`out_1` = **camera_takepic**()
INPUTS:	None.
OUTPUTS:	`out_1` – 349 × 288 matrix containing downloaded data from Peeping ROM's memory.
DESCRIPTION:	`camera_takepic` is a wrapper that calls a number of low-level functions to capture an image, download it, time the download, and close any open ports. This function also performs the resize operation that converts the downloaded datastream into a properly sized matrix for display.
EXAMPLES:	Typical usage – cause image capture and download. `camera_takepic;`
SEE ALSO:	sec 3.2.3

```
function raw_data = camera_takepic()
    %% performs picture capture and download
    %% returns matrix with raw image info, post resizing
    %% into proper 349x288 image

    %% capture picture
    disp('Starting image capture...');
    pr_picture;
    disp('Image captured.');

    %% download image and resize
    disp('Downloading data...');
    t_start = clock;
    raw_data = reshape(pr_download(349*288),349,288)';
    download_time = etime(clock, t_start);
    disp(sprintf('Total download time: %d seconds.',[round(download_time)]));

    %% now close serial port
    pr_close;
```

B.6 pr_bitset.m

NAME: `pr_bitset` – high-level bit setting function

CALL: **pr_bitset**(`arg_1`, `arg_2`, `arg_3`, [`arg_4`])

INPUTS: `arg_1` – register address to change; acceptable values from 0 to 255. If given a character string (*i.e.* `'D7'`) the input will be interpreted as a hexadecimal value.
`arg_2` – bit address to change; acceptable values from 0 (LSB) to 7 (MSB).
`arg_3` – value to set bit with; acceptable values either 0 or 1.
`arg_4` – *optional* output mode.

Argument	Effect
`'verbose'`	Causes all internal `pr_regset` calls to have `'verbose'` argument, causing detailed output for each register change.
`'quiet'`	**default mode;** `pr_modeset` does not provide detailed output to screen.

OUTPUTS: None.

DESCRIPTION: `pr_bitset` is a function designed to simplify single-bit operations for the user. This is useful in Peeping ROM since many registers are multiplexed with specific bits having unrelated purposes. It can set any bit in the camera registers to either 0 or 1 with the option of a verbose output showing all of the individual register reads and writes.

EXAMPLES: Typical usage – setting bit 4 of register `'2C'` to 0 with verbose output.

```
pr_bitset('2C',4,0,'verbose');
```

Typical usage – setting bit 7 of register 13 to 1 with no output.

```
pr_bitset(13,7,1);
```

SEE ALSO: sec 3.2.2, pr_regset, pr_set

```
function data = pr_bitset(register,bit,value,mode)
    %% bitwise operations on register
    %% call: pr_regset(<reg>,<0-index-bit#>,<value>,['verbose'/'quiet'])
    %% option for verbose output:
    %% <initial value, value written, new value>
    %% also notifies if written value isn't read back

    %% hex->dec conversion information
    %% if input is character string, convert hex to dec
    %% otherwise, decimal input
    if (ischar(register))
        register = hex2dec(register);
    end

    %% actual bit setting, etc
    if (nargin < 3)
        error('Not enough information to set register!');
    elseif (nargin == 4)
        old_value = pr_get(register);
        new_value = bitset(old_value,(bit+1),value);
        pr_regset(register,new_value,mode);
    else
        mode = 'quiet';
        old_value = pr_get(register);
        new_value = bitset(old_value,(bit+1),value);
        pr_regset(register,new_value,mode);
    end
```

B.7 pr_check.m

NAME: `pr_check` – internal camera detect function

CALL: **pr_check**()

INPUTS: None.

OUTPUTS: None.

DESCRIPTION: `pr_check` is an internal function that communicates with Peeping ROM's PIC and camera to ensure that hardware with valid firmware is responding to PC commands. It will return an error if the camera or host PIC controller is not responding, or if the hardware's firmware version is incorrect.

EXAMPLES: Typical usage – check for responsive camera.

```
pr_check;
```

SEE ALSO: n/a

```
function pr_check(verbose)

  if(nargin < 1)
    verbose = 1;
  end

  global pr_serialobj;
  pr_flush;

  %% If no format string is specified, '%s\n' is assumed.
  %% Specify it manually to avoid confusion.
  fprintf(pr_serialobj, '%s\n', 'V');
  line = fgetl(pr_serialobj);
  if(strncmp(line,'ver ',4) == 0)
    pr_close;
    error 'Invalid or no response from camera. Check connections.';
  end
  if(strcmp(line,'ver 02') == 0)
    pr_close;
    error('Wrong camera firmware version "%s", expected "ver 02"',line);
  end
  if(verbose)
          disp('Camera firmware 02 detected.');
  end
```

B.8 pr_close.m

NAME: `pr_close` – internal serial port closing function

CALL: **pr_close**()

INPUTS: None.

OUTPUTS: None.

DESCRIPTION: `pr_close` is an internal function used to close a serial port opened using `pr_init`.

EXAMPLES: Typical usage – close serial port.

```
pr_close;
```

SEE ALSO: pr_init

```
function pr_close
  clear global pr_serialobj;
  global pr_serialobj;

  ports=instrfind;
  if isobject(ports)
    stopasync(ports);
    fclose(ports);
    delete(ports);
    disp('Closed serial port.');
  end
```

B.9 pr_display.m

NAME: `pr_display` – internal figure generation function

CALL: **pr_display**(`arg_1`, [`arg_2`, `arg_3`])

INPUTS: `arg_1` – data matrix to be displayed; either M × N × 1 or M × N × 3.
`arg_2` – *optional* string containing figure name to be displayed in title bar of window; will default to "Image" if not specified.
`arg_3` – *optional* figure number to use; will default to 4321 if not specified.

OUTPUTS: None.

DESCRIPTION: `pr_display` is an internal function that takes a data matrix and creates an appropriate figure window for it. It allows for specifying the figure window name and ID number, performs 1:1 resizing, and colorizes an image if necessary. If the figure number of an existing window is supplied, the new image will refresh in the existing window rather than spawning a new one. Also, if the image data in `arg_1` is a single-plane matrix (M × N × 1) it will be rendered in grayscale with 0 being black and 255 being white. All other image data will be rendered in color with the first layer being red, the second green, and the third blue.

EXAMPLES: Typical usage – display matrix `raw_data` in figure 1001 with name "Raw Data Image".

```
pr_display(raw_data,'Raw Data Image', 1001);
```

SEE ALSO: pr_showpics

```
function pr_display(data, fig_name, fig_number)

  %% Display a peeping rom image in figure window fig_number (or 4321)
  %% Pass either a NxM matrix to display as greyscale, or
  %% a NxMx3 matrix to display as red, green, blue.
  %% also pass figure name as string

  [ row, col, planes ] = size(data);
  if planes ~= 1 && planes ~= 3
    error "Bad image data format"
  end

  %% Bring figure to front, display image, and resize
  if (nargin ~= 3)
      fig = figure(4321);
  elseif (nargin == 3)
      fig = figure(fig_number);
  else
      warning('No figure name or number passed to pr_display.');
      fig_name = 'Image';
      fig = figure(4321);
  end

  clf;
  image(uint8(data));
  if planes == 1
    %% Display as grayscale
    colormap(gray(256));
  end
  rect = get(fig, 'Position');
  set(fig, 'Position', [rect(1), rect(2), col, row]);
  set(fig, 'Resize', 'on');
  set(fig, 'Name', sprintf('Peeping ROM - %s',[fig_name]));
  set(fig, 'NumberTitle', 'off');
  ha = gca;
  set(ha, 'Units', 'Normalized');
  axis image;
  axis off;
  set(ha, 'Position', [0, 0, 1, 1]);
```

B.10 pr_download.m

NAME: `pr_download` – internal image download function

CALL: `out_1` = **pr_download**(`arg_1`, `arg_2`, `arg_3`)

INPUTS: `arg_1` – number of bytes to be downloaded; acceptable values between 0 and 512KB.
`arg_2` – *optional* RAM address to start download at.
`arg_3` – *optional* verbose flag to cause error data printing on-screen.

OUTPUTS: `out_1` – `arg_1` × 1 matrix containing all of the downloaded data, first byte received at $(1, 1)$.

DESCRIPTION: `pr_download` is an internal function that performs all of the software actions needed to download image data from Peeping ROM's SRAM. To prevent missed data frames this function requests data in 128-byte packets. Every 5 seconds the percentage downloaded is displayed in Matlab. It will return an error if an invalid number of databytes is given, if port setup fails, or if there is an error while downloading the data.

EXAMPLES: Typical usage – download 349 × 288 bytes of image data with verbose flag.

```
raw_data = pr_download(349*288,0,'verbose');
```

`raw_pic` now contains 100,512 bytes of data in a 101,512 × 1 matrix.

SEE ALSO: camera_takepic, pr_dlchunk, pr_dlsetup

```
function data = pr_download(num, addr, mode)

  %% Check arguments
  if(nargin < 1 || num < 1 || num > 512*1024 || num ~= floor(num))
    error 'Invalid number of bytes.';
  end

  if(nargin < 2)
    addr = 0;
  end

  if(nargin < 3)
    mode = 'quiet';
  end

  if(addr < 0 || addr >= 512*1024 || addr ~= floor(addr))
    error 'Invalid starting address.';
  end

  global pr_serialobj;
  pr_flush;
  pr_dlsetup(addr);
  tic;
  chunksize = 128; % must match PIC chunk size

  %% Download the data in chunks
  data = zeros(num,1);
  for start = 1:chunksize:num
    %% Print status every 5 seconds
    if toc > 5
      disp(sprintf('%6.2f%% complete',start/num * 100));
      tic
    end
    count = chunksize;
    bytes = pr_dlchunk(count, addr + start − 1, mode);
    if (start + 128) > num
      count = num − start + 1;
    end
    data(start:(start + count − 1)) = bytes(1 : count);
  end
  disp(sprintf('%6.2f%% complete',100));

  %% Exit download mode
  fprintf(pr_serialobj, '%s\n', 'X');
  line = fgetl(pr_serialobj);
```

B.11 pr_dlchunk.m

NAME: `pr_dlchunk` – packet download and error correction function

CALL: `out_1` = **pr_dlchunk**(`arg_1`, `arg_2`, `arg_3`)

INPUTS: `arg_1` – packet size (in data bytes) to receive.
`arg_2` – starting address of the data packet in RAM.
`arg_3` – *optional* verbose flag to display any serial reception errors.

OUTPUTS: `out_1` – matrix of size `arg_1`$\times 3 - 1$ containing downloaded line of data.

DESCRIPTION: `pr_dlchunk` is an internal function that downloads a small packet of the total image data. It contains error correction routines that will redownload data if it is corrupted with a frameshift error.

EXAMPLES: Typical usage – download 128 databytes of data from address 0 onwards.

```
bytes = pr_dlchunk(128,0,'verbose');
```

SEE ALSO: pr_download, pr_dlsetup

```
function bytes = pr_dlchunk(size, addr, mode)
  %% Download a chunk of specified size, retrying if there's an error.
  %% Assumes the download has been initialized, and reinitializes to
  %% the given address only if it needs to retry.

  if(nargin < 3)
    mode = 'quiet';
  end

  if(size < 1)
    error 'Invalid chunk size.';
  end
  linesize = size * 3 - 1;

  if(addr < 0 || addr >= 512*1024 || addr ~= floor(addr))
    error 'Invalid starting address.';
  end

  global pr_serialobj;
  pr_flush;

  bad = 1;
  while bad
    bad = 0;
    fprintf(pr_serialobj, '%s\n', 'M');
    line = fgetl(pr_serialobj);
    if(length(line) ~= linesize)
      if(mode == 'verbose')
        disp(sprintf('Warning: only got %d characters, wanted %d', ...
                     length(line), linesize));
      end
      bad = 1;
    else
      [bytes, count] = sscanf(line, '%02x');
      if(count ~= size)
        if(mode == 'verbose')
          disp(sprintf('Warning: only got %d data bytes, wanted %d', ...
                       count, size));
        end
        bad = 1;
      end
    end
    if(bad)
      %% reinitialize download
      if(mode == 'verbose')
        disp(sprintf('Retrying download at address %06x',addr));
      end
      pr_check(0);
      pr_dlsetup(addr);
    end
  end
```

B.12 pr_dlsetup.m

NAME: `pr_dlsetup` – RAM address setup function

CALL: **pr_dlsetup**(`arg_1`);

INPUTS: `arg_1` – RAM address to be set.

OUTPUTS: None.

DESCRIPTION: `pr_dlsetup` is an internal function that sets the address counter of Peeping ROM's SRAM to the value `arg_1`. Since this process involves incrementing the RAM to the proper address a high value in `arg_1` may take a short amount of time to complete. It also commands the PIC controller to prepare for the bulk downloading of data from SRAM. It will generate an error if the download command fails.

EXAMPLES: Typical usage – set RAM address to 1234.

```
pr_dlsetup(1234);
```

SEE ALSO: pr_download, pr_dlchunk

```
function pr_dlsetup(addr)
  %% Initialize download at specified address

  if(addr < 0 || addr >= 512*1024 || addr ~= floor(addr))
    error 'Invalid starting address.';
  end

  global pr_serialobj;
  pr_flush;

  %% Setup the download
  cmd = sprintf('D%06x', addr);
  fprintf(pr_serialobj, '%s\n', cmd);
  line = fgetl(pr_serialobj);
  if(strcmp(line,'ok') == 0)
    error 'Download setup failed.';
  end
```

B.13 pr_flush.m

NAME: `pr_flush` – internal serial buffer flushing function

CALL: **pr_flush**()

INPUTS: None.

OUTPUTS: None.

DESCRIPTION: `pr_flush` is an internal Peeping ROM function used to empty the serial input buffer before sending a new command. It will return an error if no port is open.

EXAMPLES: Typical usage – clearing port buffer.

```
pr_flush;
```

SEE ALSO: pr_init

```
function pr_flush()

  global pr_serialobj;

  if ~isobject(pr_serialobj)
    error 'Port not open.';
  end

  while pr_serialobj.BytesAvailable > 0
    c = fscanf(pr_serialobj,'%c',1);
  end
```

B.14 pr_get.m

NAME: `pr_get` – high-level register reading function

CALL: `out_1` = **pr_get**(`arg_1`)

INPUTS: `arg_1` – register address to be read. Accepts value between 0 and 255. If input is a character string (*i.e.* 'F3'), string is interpreted as hexadecimal.

OUTPUTS: `out_1` – value read from register at address `arg_1`.

DESCRIPTION: `pr_get` is a user-accessible function that reads the value of any register in Peeping ROM's camera. It will return an error if the read fails.

EXAMPLES: Typical usage – read value of register 37 (in hex, '25')

```
foo = pr_get('25');
```

The variable `foo` now contains the value of register 37.

SEE ALSO: pr_regset, sec 3.2.2

```
function val = pr_get(reg)

  %% convert hex if necessary
  if(ischar(reg))
     reg = hex2dec(reg);
  end

  if(reg < 0 || reg > 255)
    error 'Invalid register.';
  end

  global pr_serialobj;
  pr_flush;

  fprintf(pr_serialobj, 'G%02x\n', reg);
  line = fgetl(pr_serialobj);
  [val, ok] = sscanf(line, '%02x');
  if(ok ~= 1)
    pr_close;
    error 'Register read failed.';
  end
```

B.15 pr_init.m

NAME:	`pr_init` – internal serial port initialization
CALL:	**pr_init**(`arg_1`)
INPUTS:	`arg_1` – *optional* character string of serial port name, *i.e.* `'COM2'` or `'/dev/tty0'`.
OUTPUTS:	None.
DESCRIPTION:	`pr_init` is an internal function that performs automatic serial port detection and setup. If a port is not specified as `arg_1`, it will determine the computer OS and attempt to open the default serial port. In the event that Matlab cannot open that port, it will prompt the user to input a port name. Serial format is 115.2kbps, 8N1 (8 data bits, no parity, 1 stop bit).
EXAMPLES:	Typical usage – use serial port autodetect. `pr_init;` Typical usage – serial port ID is COM3. `pr_init('COM3');`
SEE ALSO:	camera_setup

```
function pr_init(port)

  %% Check arguments
  if (nargin ~= 1)
      %% no port specified, so guess based on OS
      os_type = computer;
      switch os_type
          case 'PCWIN'
              disp('No serial port specified and MS Windows detected; attempting COM1...');
              port = 'COM1';
          case 'MAC'
              serialport = input('Macintosh OS detected; please specify serial port:');
              port = serialport;
          case {'GLNX86','GLNXA64','GLNXI64'}
              disp('No serial port specified and Linux detected; attempting /dev/ttyS0...');
              port = '/dev/ttyS0';
          otherwise
              serialport = input('Unknown OS detected; please specify serial port:');
              port = serialport;
      end
  end

  pr_close;

  global pr_serialobj;
  pr_serialobj=serial(port, 'BaudRate', 115200, 'DataBits', 8, ...
                      'Parity', 'none', 'StopBits', 1, 'Terminator', 'LF', ...
                      'Timeout', 10);
  fopen(pr_serialobj);
  disp('Opened serial port.');

  pr_check;
```

B.16 pr_modeset.m

NAME: `pr_modeset` – high-level automatic mode configuration function

CALL: **pr_modeset**(`arg_1`, `arg_2`, `arg_3`, `arg_4`, `arg_5`, `arg_6`, [`arg_7`])

INPUTS: `arg_1`, `arg_3`, `arg_5` – automode options (any order).

Argument	System
`'AWB'`	Automatic White Balancing. If enabled, controls register `0x06 (BRT)`.
`'AEC'`	Automatic Exposure Control. If enabled, controls register `0x10 (AEC)`.
`'AGC'`	Automatic Gain Control. If enabled, controls registers `0x00 (GAIN)`, `0x01 (BLUE)`, `0x02 (RED)`.

`arg_2`, `arg_4`, `arg_6` – mode options.

Argument	Effect
`'enabled'`	**default mode;** Causes automatic system to change camera registers (overwriting user changes) and effect image qualities.
`'disabled'`	Deactivates internal circuits of specified system. Effected registers can now be altered by user without fear of internal overwrites.

`arg_7` – *optional* output mode.

Argument	Effect
`'verbose'`	Causes all internal `pr_regset` calls to have `'verbose'` argument, causing detailed output for each register change.
`'quiet'`	**default mode;** `pr_modeset` does not provide detailed output to screen.

OUTPUTS: None.

DESCRIPTION: `pr_modeset` is a high-level function that abstracts the user from having to manually set each register to correctly deactivate all of the automatic modes of Peeping ROM's camera. It also provides a verbose output, if desired, so all of the register read and write operations can be seen in Matlab. It will return an error if incorrect arguments are given or an incorrect number of arguments is given.
`pr_modeset` internal register operations for each mode:

Mode	Register Changes
`AWB`	Register `0x12 (COMA)`, bit 2: enabled if 1.
`AEC`	Register `0x29 (COMI)`, bit 7: enabled if 0.
`AGC`	Register `0x12 (COMA)`, bit 5: enabled if 1. Register `0x28 (COMH)`, bit 3: enabled if 0. **Note:** both conditions must be met if AGC is enabled.

EXAMPLES: Typical usage – auto white balance and auto exposure modes on, auto gain control off, verbose output.

```
pr_modeset('AWB', 'enabled', 'AEC', 'enabled', 'AGC',
                                 'disabled', 'verbose');
```

Typical usage – all modes off and verbose output supressed.

```
pr_modeset('AGC', 'disabled', 'AWB', 'disabled', 'AEC',
                                 'disabled', 'quiet'); or
pr_modeset('AGC', 'disabled', 'AWB', 'disabled', 'AEC',
                                              'disabled');
```

SEE ALSO: pr_regset, camera_usermode, sec 3.2.2

```
function data = pr_modeset(mode_1,state_1,mode_2,state_2,mode_3,state_3,mode)
    %% Sets up entire automode system and gives
    %% back useful info to the operator.
    %% Expects variable number of arguments of type
    %% <system>,<status> --> ...'AEC','disable',...
    %% AEC, AWB, AGC options

    % starts with all modes enabled!
    disp('NOTE: all automatic modes are enabled on startup!');

    for i = 1:(nargin/2)
        %% sets right arguments into variables for testing
        if (nargin < 7)
            mode = 'quiet';
        end
        if ((nargin > 7) | ((nargin ~= 7) & (mod(nargin,2) ~= 0)))
            error('%d arguments supplied to pr_modeset; 3 argument PAIRS expected.',nargin);
        elseif (i == 1)
            x = mode_1;
            y = state_1;
        elseif (i == 2)
            x = mode_2;
            y = state_2;
        elseif (i == 3)
            x = mode_3;
            y = state_3;
        end
        %% parses arguments and sets camera
        %% NOTE! bitset(..) is 1-indexed, while OV6630 datasheet is
        %% 0-indexed!
        switch x
            case 'AEC'
                %% register 0x29: COMI[7] is AEC DISABLE if 1
                switch y
                    case 'enabled' %COMI[7] = 0
                        old_value = pr_get(hex2dec('29'));
                        new_value = bitset(old_value,8,0);
                        pr_regset('29',new_value,mode);
                        disp('Auto Exposure Control operational.');
                    case 'disabled' %COMI[7] = 1
                        old_value = pr_get(hex2dec('29'));
                        new_value = bitset(old_value,8,1);
                        pr_regset('29',new_value,mode);
                        disp('Auto Exposure Control disabled.');
                    otherwise
                        error('%s is not an allowed option.',y);
                end
            case 'AWB'
                switch y
                    %% register 0x12: COMA[2] is AWB ENABLE if 1
                    case 'enabled' %COMA[2] = 1
                        old_value = pr_get(hex2dec('12'));
                        new_value = bitset(old_value,3,1);
                        pr_regset('12',new_value,mode);
                        disp('Auto White Balance operational.');
                    case 'disabled' %COMA[2] = 0
                        old_value = pr_get(hex2dec('12'));
                        new_value = bitset(old_value,3,0);
                        pr_regset('12',new_value,mode);
                        disp('Auto White Balance disabled.');
                    otherwise
                        error('%s is not an allowed option.',y);
                end
            case 'AGC'
                switch y
                    %% register 0x12: COMA[5] is AGC ENABLE if 1
                    %% register 0x28: COMH[3] is AGC DISABLE if 1
                    case 'enabled' %COMA[5] = 1; COMH[3] = 0
                        old_value = pr_get(hex2dec('12'));
                        new_value = bitset(old_value,6,1);
                        pr_regset('12',new_value,mode);
                        old_value = pr_get(hex2dec('28'));
                        new_value = bitset(old_value,4,0);
                        pr_regset('28',new_value,mode);
                        disp('Auto Gain Control operational.');
                    case 'disabled' %COMA[5] = 0; COMH[3] = 1
                        old_value = pr_get(hex2dec('12'));
                        new_value = bitset(old_value,6,0);
                        pr_regset('12',new_value,mode);
                        old_value = pr_get(hex2dec('28'));
                        new_value = bitset(old_value,4,1);
                        pr_regset('28',new_value,mode);
                        disp('Auto Gain Control disabled.');
                    otherwise
                        error('%s is not an allowed option.',y);
                end
            otherwise
                error('%s is not an allowed option.',x);
            end
        end
```

B.17 pr_picture.m

NAME: `pr_picture` – internal image capture function

CALL: **pr_picture**()

INPUTS: None.

OUTPUTS: None.

DESCRIPTION: `pr_picture` is an internal function that sends the "capture next available frame" command string to the PIC host controller. It will return an error if the PIC fails to capture an image successfully.

EXAMPLES: Typical usage – capture next image.

```
pr_picture;
```

SEE ALSO: camera_takepic

```
function pr_picture()

  global pr_serialobj;
  pr_flush;

  fprintf(pr_serialobj, '%s\n', 'P');
  line = fgetl(pr_serialobj);
  if(strcmp(line,'ok') == 0)
    pr_close;
    error 'Capture failed.';
  end
  disp('Captured picture.');
```

B.18 pr_regset.m

NAME: `pr_regset` – high-level register setting function

CALL: **pr_regset**(`arg_1`, `arg_2`, [`arg_3`])

INPUTS: `arg_1` – register to be changed. If given a string input (*i.e.* 'B7') `pr_regset` will interpret this as a hexadecimal value; otherwise will interpret as a decimal value.
`arg_2` – value to be set in register. Like `arg_1`, character string input will be interpreted as hexadecimal.
`arg_3` – *optional* argument that changes output mode of `pr_regset`. Options include:

Value	Output Behavior
`'verbose'`	Causes text output in Matlab terminal showing all register operations performed during register value set.
`'quiet'`	**default mode;** No text output is shown in Matlab unless an error is detected.

This argument is not necessary for proper operation, however, and can be left out during the function call.

OUTPUTS: None.

DESCRIPTION: `pr_regset` is the register modification function provided so the user can alter any of the camera registers. It allows multiple input formats and a verbose output of the form

```
Register <reg number>:  originally <old val>, set to <new
            val>, new value <readback val>.
```

Regardless of mode setting, text output will occur if the value sent to the register is not identical to the value read back after the write operation occurs. However, since this behavior is expected for some registers this warning will not cause an image capture to fail.

EXAMPLES: Typical usage – change register '2E' to 63 while seeing verbose output.

```
pr_regset('2E',63,'verbose');
```

SEE ALSO: sec 3.2.2, pr_modeset, pr_set

```
function data = pr_regset(register, value, mode)
    %% sets register to specific value
    %% call: pr_regset(<reg>,<value>,['verbose'/'quiet'])
    %% option for verbose output:
    %% <initial value, value written, new value>
    %% also notifies if written value isn't read back

    %% hex−>dec conversion information
    %% if register is characters, hex2dec it
    %% otherwise, decimal input
    if (ischar(register))
        register = hex2dec(register);
    end
    %% same with value argument
    if (ischar(value))
        value = hex2dec(value);
    end

    %% actual register setting, etc
    if (nargin < 2)
        error('Not enough information to set register!');
    elseif (nargin == 3)
        switch mode
            case 'verbose'
                %% verbose mode
                old_value = pr_get(register);
                pr_set(register, value); %changes register
                new_value = pr_get(register);
                %%now display the information
                disp(sprintf('Register 0x%02X: originally %02d, set to %02d, new value %02d', [register,
                    old_value, value, new_value]));
            case 'quiet'
                %% not verbose, just warn on bad readback
                old_value = pr_get(register);
                pr_set(register, value); %changes register
                new_value = pr_get(register);
                %% warn on bad readback
                if (value ~= new_value)
                    warning('Register 0x%02X set to %02d but reads %02d. Expected?', register, value,
                        new_value);
                end
            otherwise
                warning('%s passed as argument to pr_regset; using verbose mode.',mode);
                %% verbose mode
                old_value = pr_get(register);
                pr_set(register, value); %changes register
                new_value = pr_get(register);
                %%now display the information
                disp(sprintf('Register 0x%02X: originally %02d, set to %02d, new value %02d', [register,
                    old_value, value, new_value]));
        end
    else
        %% not verbose, just warn on bad readback
        old_value = pr_get(register);
        pr_set(register, value); %changes register
        new_value = pr_get(register);
        %% warn on bad readback
        if (value ~= new_value)
            warning('Register 0x%02X set to %02d but reads %02d. Expected?', register, value, new_value);
        end
    end
```

B.19 pr_regread.m

NAME:	`pr_regread` – high-level register value display function
CALL:	**pr_regread**(`arg_1`)
INPUTS:	`arg_1` – register address to be displayed; if character string is given it is interpreted as hexadecimal.
OUTPUTS:	None.
DESCRIPTION:	`pr_regread` is a function that prints the value of a register in the Matlab terminal.
EXAMPLES:	Typical usage – display value of register 'B2'. `pr_regread('2E');`
SEE ALSO:	camera_usermode

```
function pr_regread(reg)
%% register value reporting function

    %% hex−>dec conversion information
    %% if input is character string, convert hex to dec
    %% otherwise, decimal input
    if(ischar(reg))
        reg = hex2dec(reg);
    end

    %% now display the register value
    disp(sprintf('Register 0x%02X = %02d.',[reg, pr_get(reg)]));
```

B.20 pr_reinit.m

NAME: `pr_reinit` – internal camera reinitialization function

CALL: **pr_reinit**()

INPUTS: None.

OUTPUTS: None.

DESCRIPTION: `pr_reinit` causes the camera to reset. It is used once per image capture to ensure the camera is in a stable state before applying user changes and recording a photo.
Note: reinitialization causes all registers to revert back to initial values specified in `reg.h`.

EXAMPLES: Typical usage – cause camera reinitialization.

```
pr_reinit;
```

SEE ALSO: reg.h, camera_setup

```
function pr_reinit()

  global pr_serialobj;
  pr_flush;

  fprintf(pr_serialobj, '%s\n', 'R');
  line = fgetl(pr_serialobj);
  if(strcmp(line,'ok') == 0)
    pr_close;
    error 'Camera reinitialization failed.';
  end
  disp('Camera reinitialized.');
```

B.21 pr_set.m

NAME: `pr_set` – internal register value setting function

CALL: **pr_set**(`arg_1`, `arg_2`)

INPUTS: `arg_1` – register to be set; value from 0 to 255 allowed.
`arg_2` – value register will be set to.

OUTPUTS: None.

DESCRIPTION: `pr_set` communicates with the PIC to set a specified register to a particular value. It will cause an error if readback from the camera fails or an invalid register number is given.
Note: this function is not intended for use by the user; please use the improved function `pr_regset` instead.

EXAMPLES: Typical usage – setting register 24 to 128.

```
pr_set(24,128);
```

SEE ALSO: pr_regset

```
function val = pr_set(reg, setval)

  if(reg < 0 || reg > 255)
    error 'Invalid register.';
  end

  global pr_serialobj;
  pr_flush;

  fprintf(pr_serialobj, 'S%02x%02x\n', [reg; setval]);
  line = fgetl(pr_serialobj);
  [val, ok] = sscanf(line, '%02x');
  if(ok ~= 1)
    pr_close;
    error 'Register readback failed.';
  end
```

B.22 pr_showpics.m

NAME:	`pr_showpics` – high-level multi-image display function
CALL:	**pr_showpics**(`arg_1`, `arg_2`, `arg_3`, `arg_4`, `arg_5`)
INPUTS:	`arg_1` – M × N × 1 matrix containing integers 0 through 255 that will be displayed as the raw camera data. `arg_2` – M × N × 3 matrix containing integers 0 through 255; data will be displayed as the final interpolated color image with red channel being $(:, :, 1)$, green being $(:, :, 2)$, blue being $(:, :, 3)$. `arg_3` – M × N × 1 matrix containing integers 0 through 255 that will be displayed as the raw red channel data. `arg_4` – M × N × 1 matrix containing integers 0 through 255 that will be displayed as the raw green channel data. `arg_5` – M × N × 1 matrix containing integers 0 through 255 that will be displayed as the raw blue channel data.
OUTPUTS:	None.
DESCRIPTION:	`pr_showpics` generates all of Peeping ROM's output images. It calls `pr_display` for actual figure generation. It also specifies figure numbers for each image so any subsequent pictures taken with Peeping ROM will refresh into those windows rather than spawning more.
EXAMPLES:	See `picture_me` or `automode` for usage examples.
SEE ALSO:	picture_me, automode, pr_display, sec 3.5

```
function data = camera_showpics(raw_image, color_image, red_raw, green_raw, blue_raw)
    %% shows 5 photos for each picture taken:
    %% raw-red, raw-green, raw-blue,
    %% bayer-red, bayer-green, bayer-blue
    %% raw-bw, bayer-bw, full-image
    %% uses figures 1000-1008
    %% assumes MxN matrix for raw, MxNx3 for color
    disp('Displaying all image data...');

    %% after interpolation using bayer
    r_raw = zeros(length(red_raw(:,1)),length(red_raw),3);
    r_raw(:,:,1) = red_raw(:,:,1);
    g_raw = zeros(length(green_raw(:,1)),length(green_raw),3);
    g_raw(:,:,2) = green_raw(:,:,1);
    b_raw = zeros(length(blue_raw(:,1)),length(blue_raw),3);
    b_raw(:,:,3) = blue_raw(:,:,1);

    %% after interpolation using bayer
    red_channel = zeros(length(color_image(:,1)),length(color_image),3);
    red_channel(:,:,1) = color_image(:,:,1);
    green_channel = zeros(length(color_image(:,1)),length(color_image),3);
    green_channel(:,:,2) = color_image(:,:,2);
    blue_channel = zeros(length(color_image(:,1)),length(color_image),3);
    blue_channel(:,:,3) = color_image(:,:,3);

    %% now display everything
    pr_display(raw_image, 'Raw Data', 1000);
    pr_display(r_raw, 'Raw Red Channel', 1001);
    pr_display(g_raw, 'Raw Green Channel', 1002);
    pr_display(b_raw, 'Raw Blue Channel', 1003);
    pr_display(red_channel, 'Post-Bayer Red Channel', 1004);
    pr_display(green_channel, 'Post-Bayer Green Channel', 1005);
    pr_display(blue_channel, 'Post-Bayer Blue Channel', 1006);
    pr_display(color_image, 'Final Interpolated Image', 1007);
```

B.23 bayer.m

NAME: `bayer` – high-level bayer pattern interpolation function

CALL: [`out_1 out_2 out_3 out_4 out_5 out_6`] = **bayer**(`arg_1`)

INPUTS: `arg_1` – an M × N × 1 matrix containing integers from 0 to 255; typically raw data from camera.

OUTPUTS: `out_1` – M × N matrix containing interpolated data from red pixel locations in bayer pattern; values from 0 to 255. Becomes red channel of final image.
`out_2` – M × N matrix containing interpolated data from green pixel locations in bayer pattern; values from 0 to 255. Becomes green channel of final image.
`out_3` – M × N matrix containing interpolated data from blue pixel locations in bayer pattern; values from 0 to 255. Becomes blue channel of final image.
`out_4` – M × N matrix containing raw data from red pixel locations in bayer pattern; values from 0 to 255. Non-red pixel locations contain value of 0.
`out_5` – M × N matrix containing raw data from green pixel locations in bayer pattern; values from 0 to 255. Non-green pixel locations contain value of 0.
`out_6` – M × N matrix containing raw data from blue pixel locations in bayer pattern; values from 0 to 255. Non-blue pixel locations contain value of 0.

DESCRIPTION: `bayer` performs a simple averaging-type bayer interpolation on a data matrix. It also provides access to all of the raw data channels (R,G,B) coming from the camera.

EXAMPLES: See `picture_me` and `automode` for usage examples.

SEE ALSO: sec 3.3, picture_me, automode

```
function [r, g, b, or, og, ob] = bayer(c)
%%% Given n x m matrix of Bayer−patterned data from camera, return three
%%% n x m matrices representing the per−pixel interpolated RGB values.
%%% All data should be integers 0−255. Interpolation is linear, and
%%% expected pattern is
%%% G B G B G B
%%% R G R G R G
%%% G B G B G B
%%% R G R G R G
%%% written 01/2005 by Jim Paris <jim@jtan.com>
  [m,n] = size(c);
  if(m<2 | n<2) error('image too small'); end;
  r = zeros(m,n);
  g = zeros(m,n);
  b = zeros(m,n);

  xO = 1:2:n; yO = 1:2:m; % all odd indices
  xE = 2:2:n; yE = 2:2:m; % all even indices
  xo = 3:2:n−1; yo = 3:2:m−1; % odd indices except border
  xe = 2:2:n−1; ye = 2:2:m−1; % even indices except border

  %% Set known RGB values from Bayer pattern
  g(yO,xO) = c(yO,xO); b(yO,xE) = c(yO,xE);
  r(yE,xO) = c(yE,xO); g(yE,xE) = c(yE,xE);

  %% Provide the uninterpolated values to the user
  or = r;
  og = g;
  ob = b;

  %%%% Do the bulk of the interpolation:

  %% Interpolate red: 2 3
  %% R 1
  r(yE,xe) = (r(yE,xe−1) + r(yE,xe+1)) / 2; % 1
  r(yo,xO) = (r(yo−1,xO) + r(yo+1,xO)) / 2; % 2
  r(yo,xe) = (r(yo−1,xe−1) + r(yo−1,xe+1) + r(yo+1,xe−1) + r(yo+1,xe+1))/4; % 3

  %% Interpolate green: G 1
  %% 2 G
  g(yo,xe) = (g(yo,xe−1) + g(yo,xe+1) + g(yo−1,xe) + g(yo+1,xe))/4; % 1
  g(ye,xo) = (g(ye,xo−1) + g(ye,xo+1) + g(ye−1,xo) + g(ye+1,xo))/4; % 2

  %% Interpolate blue: 1 B
  %% 3 2
  b(yO,xo) = (b(yO,xo−1) + b(yO,xo+1)) / 2; % 1
  b(ye,xE) = (b(ye−1,xE) + b(ye+1,xE)) / 2; % 2
  b(ye,xo) = (b(ye−1,xo−1) + b(ye−1,xo+1) + b(ye+1,xo−1) + b(ye+1,xo+1))/4; % 3

  %%%% Now fix up the edge cases on the borders:

  %% Red: copy line to top, and maybe bottom and right
  r(1,:) = r(2,:);
  if( mod(m,2)) r(m,:) = r(m−1,:); end;
  if(˜mod(n,2)) r(:,n) = r(:,n−1); end;
```

```
%% Green: average three pixels at border: G 1 G
%% This is the messy case. 2 G .
%% First do top and left G . .
g(1,xe) = (g(2,xe) + g(1,xe−1) + g(1,xe+1))/3; % 1
g(ye,1) = (g(ye,2) + g(ye−1,1) + g(ye+1,1))/3; % 2
if(mod(m,2)) % Bottom is G 1 G 1
  g(m,xe) = (g(m−1,xe) + g(m,xe−1) + g(m,xe+1))/3; % 1
else  % Bottom is 2 G 2 G 2 G
  g(m,1) = (g(m−1,1) + g(m,2))/2; % bottom left corner
  g(m,xo) = (g(m−1,xo) + g(m,xo−1) + g(m,xo+1))/3; % 2
end
if(mod(n,2)) % Right is 1 G 1 G going down
  g(ye,n) = (g(ye,n−1) + g(ye−1,n) + g(ye+1,n))/3; % 1
else % Right is G 2 G 2 going down
  g(1,n) = (g(1,n−1) + g(2,n))/2; % top right corner
  g(yo,n) = (g(yo,n−1) + g(yo−1,n) + g(yo+1,n))/3; % 1
end
if(xor(mod(m,2),mod(n,2)))
  g(m,n) = (g(m,n−1) + g(m−1,n))/2; % bottom right corner
end

%% Blue: copy line to left, and maybe bottom and right
b(:,1) = b(:,2);
if(~mod(m,2)) b(m,:) = b(m−1,:); end;
if( mod(n,2)) b(:,n) = b(:,n−1); end;

%% Make it integers again.
r = floor(r);
g = floor(g);
b = floor(b);
```

B.24 histogram.m

NAME: `histogram` – high-level brightness histogramming function

CALL: `output_1` = **histogram**(`arg_1`,`arg_2`)

INPUTS: `arg_1` – 3 Plane (M × N × 3) matrix containing integers from 0 to 255; typically the final RGB color picture.
`arg_2` – figure number of window where histogram will be displayed; typically 1008.

OUTPUTS: `out_1` – 1 × 256 matrix containing normallized frequencies of each integer (0 through 255) in composite image; values range from 0 to 1.

DESCRIPTION: `histogram` displays the frequency of each value 0 through 255 in a *composite brightness* matrix generated from `arg_1`, typically the RGB final image. It produces a graphical output of pixel brightness frequencies and returns a matrix containing the plotted data. Composite brightnesses generated by equation

$$P_{brightness} = (.299 \cdot P_{red}) + (.587 \cdot P_{green}) + (.114 \cdot P_{blue})$$

computed at each point (x, y) in M × N × 3 matrix `arg_1`.

EXAMPLES: Typical usage – generate a histogram of M × N × 3 matrix `color_pic`:

`hist` = **histogram**(`color_pic`,1008);

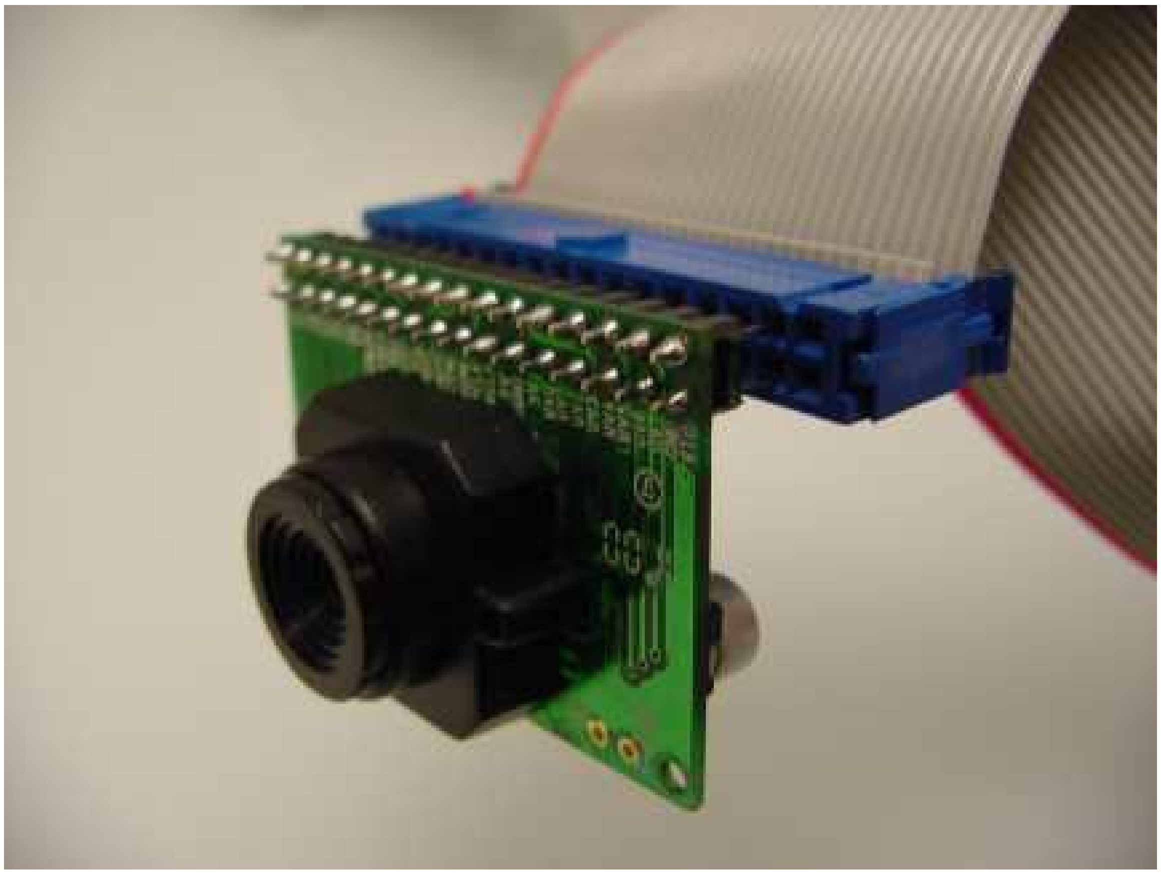
SEE ALSO: sec 3.4, picture_me, automode

```
function hist=histogram(color_img, fig_number)
    %% histogram generation and display function for PR
    %% expects RGB 3-plane color image and figure number
    %% displays histogram in specified figure

    %% create histogram matrix
    hist = zeros(1,256);

    %% get matrix size
    [n,m,p] = size(color_img);
    if (p ~= 3)
        error('Histogramming requires full RGB matrix on input.');
    end

    %% generate composite brightness matrix
    %% brightness of pixel = (.299*R)+(.587*G)+(.114*B)
    comp_brt = zeros(n,m);
    for x = 1:n;
        for y = 1:m;
            comp_brt(x,y) = round((.299*color_img(x,y,1))+(.587*color_img(x,y,2))+(.114*color_img(x,y,3))
                );
        end
    end

    %% sort by value composite brightness matrix
    for i = 0:1:255;
        s = 0;
        for x = 1:n;
            for y = 1:m;
                if (comp_brt(x,y) == i);
                    s = s+1;
                end
            end
        end
        s = s/(m*n);
        hist(1,(i+1)) = s;
    end

    %% now generate figure
    fig = figure(fig_number);
    set(fig, 'Resize', 'on');
    set(fig, 'Name', 'Peeping ROM - Image Histogram');
    set(fig, 'NumberTitle', 'off');
    axis image;
    bar(hist)
```

C Appendix C: The OmniVision OV6630 Camera Module

C.1 OV6630 Datasheet

The OmniVision 6630 Color CMOS camera is the basis of Peeping ROM. This datasheet provides all sorts of useful information on the device itself, including pinouts, timing diagrams, and register usages. Particularly useful are pages 20 through 27, which detail the exact functionality of each register accessible through Peeping ROM's Matlab interface.


OV6630 Color Camera Module