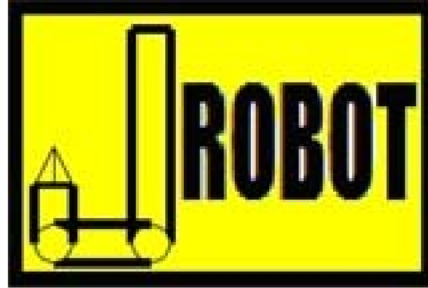


AVRcam Code Commentary

Version 1.3

Copyright © JROBOT 2007

Revision History

Date	Version	Author	Description
2/15/2007	1.0	John Orlando	Initial release
2/22/2007	1.1	John Orlando	Added sections for User Interface and Color Tracking Logic.
3/3/2007	1.2	John Orlando	Completed the Color Tracking Logic section, and added the Frame Dump section. Also added additional details throughout.
4/26/2007	1.3	John Orlando	Corrected a minor error in the Frame Dump section, where currentLineBuffer and previousLineBuffer were flipped around

References

[1] AVRcam User's Manual http://www.jrobot.net/Download.html
[2] OV6620 image sensor data sheet http://www.ovt.com

1 Introduction

1.1 Overview

The AVRcam is a small image processing system capable of performing real-time object tracking of eight different colored object simultaneously, at 50 frames/sec. This system is also capable of capturing individual snapshots of its surroundings.

1.2 Scope

This document is intended to provide a detailed description of how the AVRcam embedded software works. It is divided into logical sections covering the different aspects of the software:

- **Initialization**-This section covers the sequence of events that take place when power is first applied to the system. This includes the functionality of the Atmel AVR tiny12 on-board co-processor as well as the startup of the Atmel AVR mega8.

- **Event Loop/Dispatcher**-This section covers the basics of the event loop and corresponding infrastructure that is used to publish and process events in the system.

- **User Interface**-This section covers the user-interface provided by the AVRcam through the serial port. It includes the interface to the UART hardware, as well as the command parser responsible for processing the input.

- **Color Tracking Logic**-This section covers the processing that is done to track colorful objects. It includes the order of events for interfacing to the OV6620 image sensor to read its data bus, determine if a particular set of sampled pixels corresponds to a color of interest, and run-length encoding the packet. It also covers the processing necessary to build connected regions of colored objects, and send the results out through the user-interface.

- **Frame Dumping Logic**-This section covers the processing that is done to capture a single frame and send the corresponding image out through the user-interface.

The basic architecture of the embedded software executing on the mega8 can be found in Figure 1. It is assumed that the reader is already familiar with the general functionality of the AVRcam, as well as the hardware involved in the system. Introductory information on the AVRcam can be found in the AVRcam User's Manual located at:

www.jrobot.net/Download.html

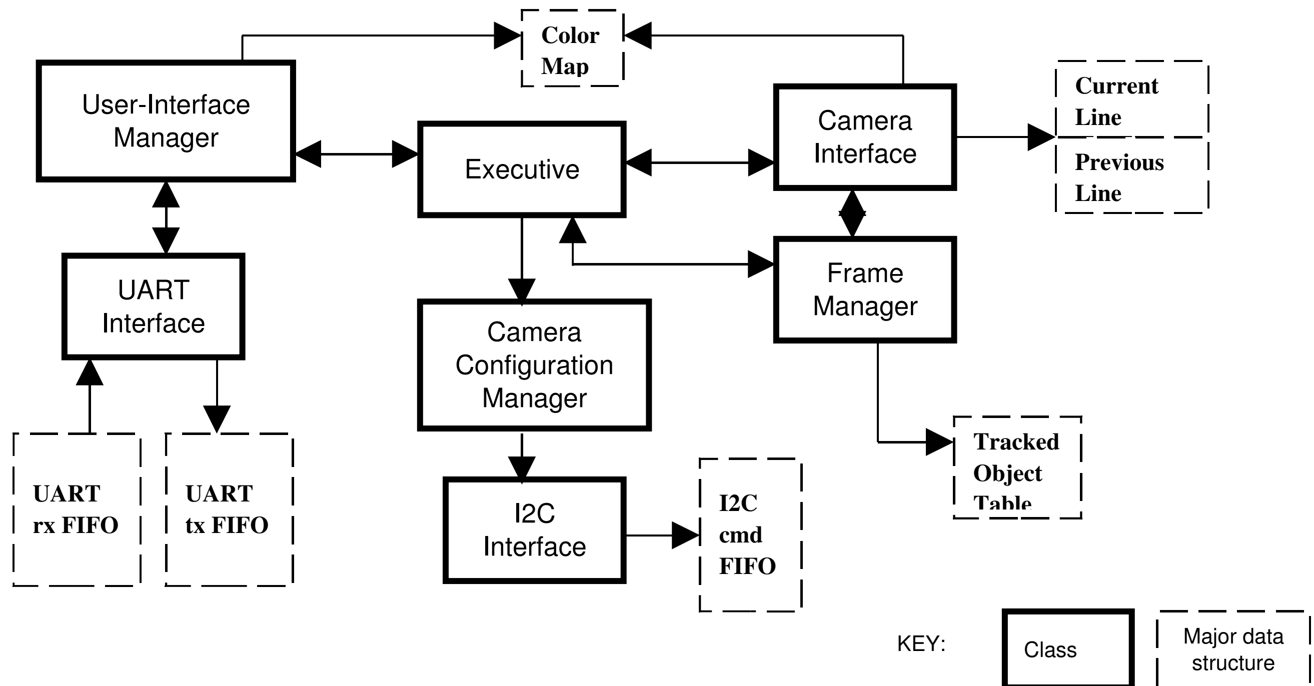


Figure 1: Software architecture of the AVRcam

2 Initialization

2.1 Files involved:

AVR tiny12 - AVRcam_tiny12.asm
AVR mega8 - main.c, FrameMgr.c, CamInterface.c, UIMgr.c, UartInterface.c, CamConfig.c, DebugInterface.c

2.2 Description

The initialization of the AVRcam actually spans across two different processors. In addition to the primary processor (AVR mega8), a small secondary processor (AVR tiny12) is also utilized. The tiny12 is responsible for some initial configuration of the system (described below) that must take place before the mega8 can start up.

2.2.1 AVR Tiny12 Initialization

At power up, the tiny12 holds the mega8 in reset by asserting the mega8's /RESET line. This provides a window of time for the tiny12 to perform two important operations. The first operation is to send an I2C command to the OV6620 image sensor to force it to tri-state its Y/UV data busses. This is necessary because the UV pixel data bus on the OV6620 interfaces to the mega8 through the same i/o lines needed for reprogramming the mega8. Thus if the user attempted to reprogram the mega8 while the OV6620 was driving its UV bus, the reprogramming operation would fail. The OV6620 image sensor's UV data bus interfaces to the mega8 through the lower four bits of PORTB. The critical line that must be tri-stated is bit 3 of PORTB, which is the MOSI line.

This tri-stating operation is completed after the tiny12 performs an I2C write to address 0x13 of the OV6620's register set, with a value of 0x05 (see [2] for the OV6620's complete register set).

The second operation handled by the tiny12 at power up is the sending of an I2C command to the OV6620 to force the image sensor to output its 17.7 MHz crystal oscillator signal on its FODD line. This output signal is available on the 32-pin header provided by the C3088 camera board, and thus can be accessed easily. But why is this necessary?

The AVRcam is designed around the concept of synchronized clocking of both the image sensor and the primary processor. In other words, the OV6620 and the mega8 need to share the same clock in order to perform the necessary streaming image processing in real-time (see the Color Tracking Logic and Frame Dump Logic sections for the details of why this is necessary).

The C3088 camera board (which utilizes the OV6620 image sensor) has a 17.7 MHz oscillator already built into it that is used as the clock source for the OV6620. To make the AVRcam work, this 17.7 MHz oscillator signal needs to be available so that the mega8 can also utilize it as its clock source. However, by default, this clock signal is not externally available on any of the header pins of the C3088. Thus, the tiny12 takes care of sending the necessary I2C command to the OV6620 so enable the gating of the oscillator signal to an output line. This output line is connected directly to the mega8's XTAL1 line. Once configured, the mega8 has the synchronized clock source that it needs.

This activity is completed after the tiny12 performs an I2C write to address 0x3F of the OV6620's register set, with a value of 0x42 (see [2] for the OV6620's complete register set).

After these two I2C write operations are completed, the tiny12 de-asserts the /RESET line of the mega8, allowing it to start up. The tiny12 then sits in an infinite loop doing nothing as the mega8 begins its processing.

2.2.2AVR mega8 Initialization

The mega8 begins execution in main.c in its main() routine. The first order of business here is to initialize the various modules utilized in the system. It first calls DebugInt_init() in DebugInt.c, which causes the yellow debug LED on the AVRcam to blink four times to indicate that the system is about to power-up. Note that the UV data bus of the OV6620 is still tri-stated at this point, due to the earlier request from the tiny12. This is to provide a four-second window of time to allow the mega8 to be re-programmed if desired by the user. Once the four blinks have completed, the DebugInt_init() routine returns.

Next, the UartInt_init() function is called in UartInt.c. This function is responsible for setting up the on-board UART to operate at 115.2 kbps, 8 data bits, 1 stop bit, no parity. This UART provides the main user interface to the system.

The initialization of the I2C interface on the mega8 occurs next, through a call to I2Cint_init() routine in I2Cint.c. This routine is responsible for setting up the on-board I2C hardware so that it is capable of acting as an I2C master utilizing the slower 100 KHz I2C timing. There is no specific reason to use 100 KHz compared to the more common 400 KHz; it simply worked right from the start and was never worth modifying.

Once the I2Cint_init() routine has completed, the CamInt_init() routine in CamInt.c is called to initialize the Camera Interface module. This routine is responsible for setting up the hardware needed to interface to the OV6620 image sensor. The interface between

the OV6620 and the mega8 consists of 12 signals (see the hardware block diagram in Figure 2):

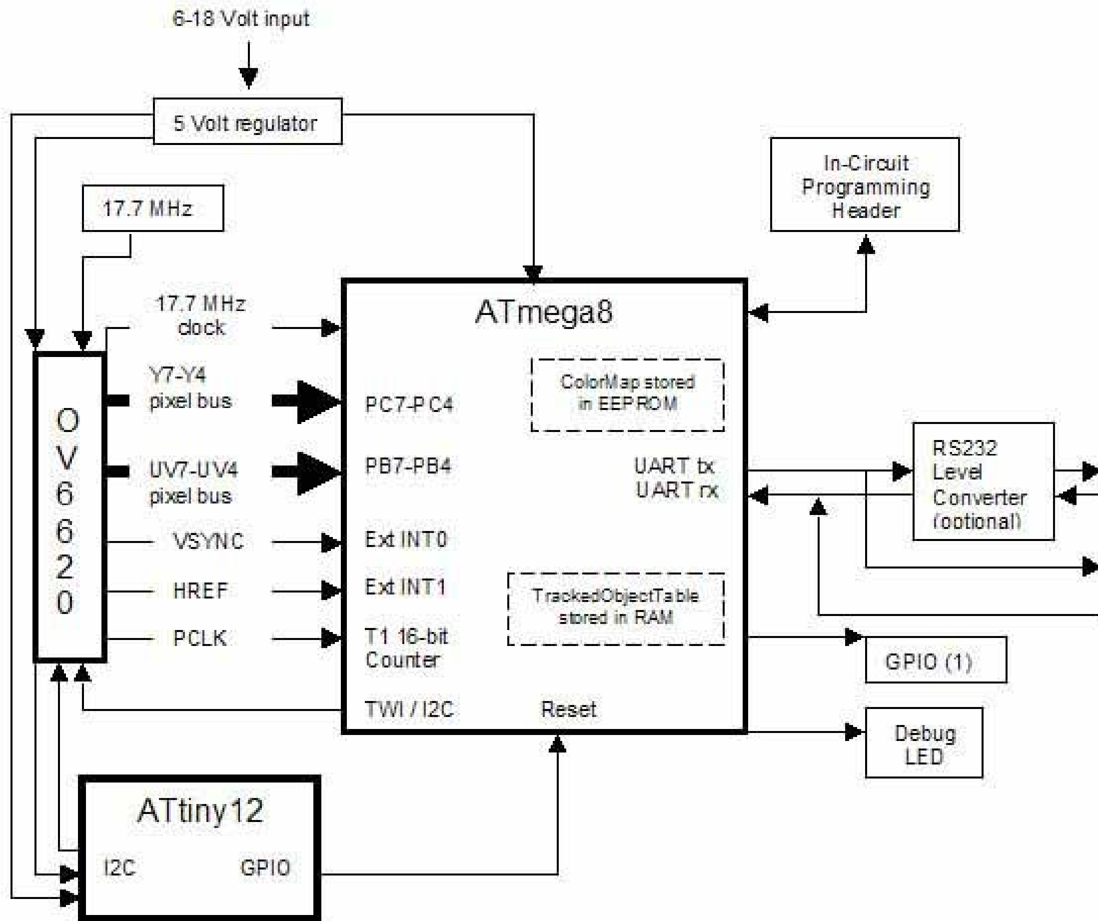


Figure 2: AVRcam Hardware Block Diagram

- **VSYNC**-This signal, short for vertical synchronization, is an output from the OV6620 sensor, indicating each time a new image frame is about to begin. It goes high and then low at the beginning of each frame. This signal is connected to pin 4 of the mega8 which is the INT0 external interrupt line. CamInt_init() configures this interrupt so that it is a rising-edge interrupt, and enables it immediately:

```
/* set up External Interrupt0 to interrupt us on rising edges
(VSYNC) */
MCUCR |= (1<<ISC01) | (1<<ISC00);/* rising edge interrupt */
GICR  |= (1<<INT0);    /* interrupt request enabled */
```

- **HREF**-This signal, short for horizontal reference, is an output from the OV6620 sensor, indicating each time a new row of pixels is about to be sent out from the camera. It goes high and then low at the beginning of each image line. This signal is connected to two input pins on the mega8. The first input is the INT1 external interrupt line. CamInt_init() configures this interrupt so that it is a rising edge interrupt, but doesn't enable it yet:

  ```
  MCUCR |= (1<<ISC11) | (1<<ISC10);  /* rising edge interrupt */
  ```

  The HREF signal is also connected to pin 6 of the mega8, which is the input to a hardware timer/counter (TIMER0). This allows the hardware timer/counter to keep track of the number of times the HREF signal toggles, and thus the count of image lines in the current frame. When configured properly, this counter will overflow and generates an interrupt after it has counted 144 pulses on the HREF line, thus indicating the end of a frame (details of this configuration can be found in the Color Tracking Logic section).

  ```
  /* set up Timer0 to count and be clocked from an external pulse
  source(HREF) on falling edges...eventually, we need to enable the
  interrupt for this!  FIX THIS */
  TCCR0 = (1<<CS02)|(1<<CS01)|(0<<CS00);
  ```

  Note: The "FIX THIS" comment in the current release references a fix that has already been implemented.

- **PCLK**-This signal, short for pixel clock, is an output from the OV6620. It is asserted when valid pixel data is available on the Y and UV data busses. When being used in the AVRcam, this corresponds to toggling 176 times per image line. This signal is connected to pin 11 of the mega8, which is the input to the hardware timer/counter TIMER1. Much like TIMER0 counts HREF pulses, TIMER1 is setup to count PCLK pulses and overflow the counter after 176 have been received. This overflow will trigger an internal counter overflow interrupt, indicating that the line of pixel data is over. TIMER1 is configured here, but the interrupt is only enabled when it is needed:

  ```
  /* ensure that timer1 is disabled to start...eventually, when PCLK
  needs to feed timer1 through the external counter, it will be
  enabled on an "as needed" basis...*/
  TCCR1B &= ~( (1<<CS12)|(1<<CS11)|(1<<CS10) );
  ```

- **Y/UV Data Busses**-These are the two 8-bit output data busses provided by the OV6620. These data busses carry the actual pixel information, synchronous with the PCLK signal. Only the top 4-bits of each data bus is

connected to the mega8 to simplify the processing. The top 4-bits of the Y bus are connected to the lower 4-bits of PORTC on the mega8, and the top 4-bits of the UV bus are connected to the lower 4-bits of PORTB on the mega8.

It is necessary to setup PORTB and PORTC so that they are inputs (Note: the AVRcam code references the Y port as the CAM_RB_BUS and the UV port as the CAM_G_BUS, since the OV6620 is used in RGB mode instead of YUV mode. The OV6620 will later be configured such that only red and blue pixel values are sent on the CAM_RB_BUS and only green pixel values are sent on the CAM_G_BUS…see the Color Tracking Logic section for more details):

```
CAM_G_BUS_DIR &= 0xF0;  /* 4-bit G bus all inputs */
CAM_G_BUS_DIR |= 0xF0;  /* disable the pull-up on PB4 and PB5 */
CAM_RB_BUS_DIR &= 0xF0;  /* 4-bit RB bus all inputs */
```

The last thing the CamInt_init() routine does is setup its colorMap table. This colorMap serves as the means to determine if a particular set of received red/green/blue pixel values sampled over the OV6620's data busses maps to one of the eight user-configured colors. This is one of the most confusing parts of how the AVRcam performs its processing, so some extra explanation is provided.

For purposes of tracking color blobs, the user must be able to provide the AVRcam with the colors of interest to track. But how are colors represented by the OV6620 camera module? Colors are represented as combinations of red, green, and blue pixels, where each pixel is an 8-bit value (in reality, the OV6620 only represents pixel values with a minimum value of 10 and a maximum value of 240, but for all practical purposes the pixels should be considered to be 8-bit values). Each image frame sent out by the OV6620 is configured as a 176 pixel by 144 pixel grid providing the following pixel arrangement:

	1	2	3	4	5	6	7	8	…	176
Line 1	G	G	G	G	G	G	G	G	G	G
Line 2	R	B	R	B	R	B	R	B	R	B
Line 3	G	G	G	G	G	G	G	G	G	G
Line 4	R	B	R	B	R	B	R	B	R	B
…										
Line 144	R	B	R	B	R	B	R	B	R	B

Figure 3: Pixel arrangement for each image frame sent from the OV6620

Note that the above pixel grid is similar to a Bayer color pattern, but slightly modified. In a typical Bayer color pattern, the odd image lines would alternate red and green pixels,

while the even image lines would alternate green and blue pixels. The OV6620 is specifically configured to this alternate Bayer formation by the AVRcam, where only green pixels are provided on the odd lines, and alternating red and blue pixels are provided on the even lines. There is a reason for this, and will be explained in the Tracking Color Logic section.

Individual colors in an image are represented by varying the values of red, green, and blue for the particular area of the color. Its possible to represent a wide variety of colors by looking at a four-pixel region comprised of red, green, and blue values (as shaded in the above table…note that there are two green values: they can either be averaged together to come to the best estimation of green for the region, or more simply, only utilize one of them and ignore the other one).

But how does a user specify a color of interest for purposes of tracking? This is accomplished by specifying a range of red, green, and blue values that should be associated with a particular color of interest. For example, a color of interest may be represented when the red pixel value is between 64 and 96, the green pixel is between 16 and 32, and the blue pixel is between 192 and 240. This would correspond to a darkish-purple color. Any time a connected region of three pixels meets the above criteria, it is considered a color of interest. So, it really comes down to accurately representing these color ranges, and then efficiently figuring out if a particular set of red/green/blue pixels maps into any of the colors of interest.

This is where the colorMap comes in. The colorMap is a 48-byte chunk of memory that should really be viewed as follows:

colorMap is 48 contiguous bytes, where the RED_MEM_OFFSET is set to 0, the GREEN_MEM_OFFSET is set to 16, and the BLUE_MEM_OFFSET is set to 32:

Bytes 0-15	Bytes 16-31	Bytes 32-47	
Red1	Green1	Blue1	Range: 0-15
Red2	Green2	Blue2	Range: 16-31
Red3	Green3	Blue3	Range: 32-47
Red4	Green4	Blue4	Range: 48-63
Red5	Green5	Blue5	Range: 64-79
Red6	Green6	Blue6	Range: 80-95
Red7	Green7	Blue7	Range: 96-111
Red8	Green8	Blue8	Range: 112-127
Red9	Green9	Blue9	Range: 128-143
Red10	Green10	Blue10	Range: 144-159
Red11	Green11	Blue11	Range: 160-175
Red12	Green12	Blue12	Range: 176-191
Red13	Green13	Blue13	Range: 192-207
Red14	Green14	Blue14	Range: 208-223
Red15	Green15	Blue15	Range: 224-239
Red16	Green16	Blue16	Range: 240-255

Color value range

For representing a single color, a single bit in each byte is assigned to represent the color in the colorMap. The AVRcam can track up to eight colors simultaneously. Each user-defined color influences only a single bit in each of the 48 bytes in the colorMap.

This is best explained by an example. To represent a single color mapped to the most-significant bit of each byte in the colorMap (for example, lets use the purple color mentioned above, represented when the red pixel value is between 64 and 96, the green pixel is between 16 and 32, and the blue pixel is between 192 and 240), the following colorMap would be setup:

0x00	0x00	0x00	Range: 0-15
0x00	0x80	0x00	Range: 16-31
0x00	0x00	0x00	Range: 32-47
0x00	0x00	0x00	Range: 48-63
0x80	0x00	0x00	Range: 64-79
0x80	0x00	0x00	Range: 80-95
0x00	0x00	0x00	Range: 96-111
0x00	0x00	0x00	Range: 112-127
0x00	0x00	0x00	Range: 128-143
0x00	0x00	0x00	Range: 144-159
0x00	0x00	0x00	Range: 160-175
0x00	0x00	0x00	Range: 176-191
0x00	0x00	0x80	Range: 192-207
0x00	0x00	0x80	Range: 208-223
0x00	0x00	0x80	Range: 224-239
0x00	0x00	0x00	Range: 240-255

Color value range

Notice that the MSB of certain bytes in the above table has been modified, since this bit is allocated to hold the red/green/blue color ranges for the first color. Each additional color to be tracked is mapped to another bit in each byte in the colorMap.

Eventually, this colorMap will serve a key purpose in the on-the-fly image processing that needs to be done to determine if a particular red/green/blue sample of pixels correlates to a user-defined color of interest. This will be discussed further in the Tracking Color Logic section.

Back to the CamInt_init() function. The colorMap is actually loaded from EEPROM located in the mega8 as the last function in the CanInt_init() routine. Each time the user configures a new colorMap, it gets written to EEPROM and is thus maintained between power cycles.

Once CamInt_init() has completed and returned, the main() routine globally enables interrupts.

Next, the CamConfig_init() routine is called in the CamConfig.c file. This routine is responsible for configuring the OV6620 image sensor. This configuration process requires the mega8 to write to several of the registers in the OV6620 via I2C commands. In particular, this routine performs the following register updates (see [2] for the complete details of the registers provided by the OV6620):

Register Address	New Value	Reason
0x14	0x20	Reduces the frame size from the default of 352x288 to 176x144.
0x39	0x40	Gate PCLK with HREF so that PCLK is only active when valid pixel data is being provided. This is needed because PCLK is feeding a hardware counter in the mega8, and we only want it counting when valid data is being received.
0x12	0x28	Change from the default YUV colorspace to RGB colorspace, and disable the auto-white balancing since it can change the hue of the received color in unexpected ways.
0x28	0x05	Setup the color sequencer in the image sensor so that the odd image lines contain just green pixel values, and the even pixel lines provide alternating red and blue pixel values.
0x13	0x01	Un-tri-state the pixel data busses to allow the pixel data to flow. These were previously tri-stated to allow for in-system re-programming of the mega8 during power up.

Finally, these commands are queued up and sent down to the OV6620 image sensor via the I2C bus. Once this is completed, the CamConfig_init() function returns.

The next module to initialize is the user-interface manager, located in UIMgr.c. The function responsible for initializing this module is UIMgr_init(). This routine resets a few of the critical variables needed for the UIMgr to function properly, and then returns.

The final module requiring initialization is the frame manager, located in FrameMgr.c. The init routine is FrameMgr_init(), and simply does some memory initialization of the trackedObjectTable data structure. This data structure is used to keep track of up to eight tracked objects. The per-tracked-object data include the color of the object, interim parameters to track the minimum and maximum x-coordinates of the tracked object, the (x,y) coordinates of the upper left corner of the object, and the (x,y) coordinates of the lower-right corner of the object.

Once FrameMgr_init() completes its initialization, a final one-second delay in main() provides a short window of time to allow the OV6620 to stabilize. After this, the system is fully initialized, and a call to Exec_run() starts the event processing loop. This loop provides the processing for all published events in the system, and doesn't ever return.

3 Event Loop and Processing

3.1 Files Involved:

AVR mega8 - Executive.c, Executive.h, Events.h

3.2 Description

The AVRcam uses a very simple event processing loop for handling events that occur in the system. This processing loop is provided by the Exec_run() routine in Executive.c. The complete enumeration of all events in the system can be found in Events.h.

The Exec_run() routine processes two different types of events. The first type of event is a "fast-event", meaning that this event must be processed as quickly as possible. These events require higher-priority processing due to the real-time requirements of setting up and processing each image line as it is streaming in from the OV6620 image sensor. There are two types of "fast events":

- **FEV_ACQUIRE_LINE_COMPLETE**: This event indicates that an image line (a total of 176 pixels) has been sampled and run-length encoded. This run-length encoded version of the image line can then be processed to determine if there are any colors present in the image line that are connected to the image line that preceded it. This processing is performed in FrameMgr_processLine(), located in FrameMgr.c In

addition, after each image line is acquired and processed, the UIMgr is given a chance to transmit a single byte out through the UART. Normally, the UIMgr will have data pending in a transmit FIFO corresponding to the tracked objects found during the previous frame. Using this method of sending one byte out at the end of processing each image line, it is possible to send up to 144 bytes out per frame (at 115.2 kbps), allowing the mega8 to continue its processing while the hardware UART performs its transmission.

- **FEV_PROCESS_LINE_COMPLETE**: This event indicates that the processing of an image line has been completed, and the system should setup to process the next line of the image. It calls the FrameMgr_acquireLine() routine which actually performs the setup.

It should be noted that there is no mechanism in place to queue up multiple "fast events". These events are logged by a simple bitfield (fastEventBitmask). The event has either occurred or not, indicated by a particular bit in fastEventBitmask being set.

The second type of event in the system is a "normal event". The majority of the events in the system are of this type. The simple executive provides an event FIFO that can queue up to eight events (defined in Events.h), and must be a power of 2. Events are "published" and added to the event FIFO through a call to Exec_writeEventFifo(), and these events are added from a variety of different spots throughout the system.

These "normal events" correspond to all sorts of different things that can happen in the system, such as receiving serial data from the user-interface UART, or having received and decoded a command from the user such as enable-tracking. Each event has a specific list of routines associated with how the event should be processed. The enumeration of the processing routines for each event in the processing loop is a crude way of providing a set of "subscribers" for the different events in the system. Each processing routine is responsible for checking the current state of the system to determine how the event should be processed.

4 User Interface

4.1 Files Involved:

AVR mega8 – UIMgr.c, UartInterface.c, Events.h

4.2 Description

The AVRcam provides a user-interface through the mega8's on-board hardware UART. This UART provides a means to receive and send serial data with an external controller. The AVRcam supports a very simple set of commands from the external controller.

These commands are sent to the AVRcam as simple carriage-return-terminated text strings, such as “GV\r”. The complete set of commands can be found in [1].

The processing provided by the user-interface is initiated each time a serial byte is received from the UART on the mega8. Whenever a serial byte is received, the SIG_UART_RECV interrupt service routine is called (located in UartInterface.c). This routine adds the received byte to the UIMgr’s receive FIFO, and then publishes an EV_SERIAL_DATA_RECEIVED event by writing directly to the executive’s event FIFO.

Once the serial receive ISR is completed, the main Exec_run() processing loop will first dispatch the EV_SERIAL_DATA_RECEIVED event to the UIMgr. The UIMgr_dispatchEvent() function checks the received event to determine what action to take. If serial data was received, it calls UIMgr_processReceivedData(). This routine is responsible for processing each incoming serial byte to determine if it is a recognized command. The actual processing depends on what character was received. Incoming characters are built into space-delimited tokens (where the incoming characters are temporarily stored in asciiTokenBuffer), and groups of tokens are built into a token list (stored in tokenBuffer).

Since all valid commands end in a ‘\r’ character, the reception of this character is the trigger to perform the command processing. The first step in this processing is to determine if the command was valid. If an invalid command was received, a negative-acknowledge message is sent to the user, and the EV_SERIAL_DATA_PENDING event is published to send out the serial data associated with this message. If the received command is valid, and acknowledge message is sent to the user, and UIMgr_executeCmd() is called to execute the command.

The UIMgr_executeCmd() routine can perform several different actions based on the received command. The following table shows the received commands and their associated actions:

Received Command	**Action**
PingCmd	Do nothing, as the already-sent ACK is sufficient
GetVersionCmd	Report the software version string programmed into the AVRcam.
ResetCameraCmd	Reset the camera (Note: This command is not currently implemented, as the resetting of the OV6620 would cause the clock signal needed by the mega8 to stop.
DumpFrameCmd	Generate the EV_DUMP_FRAME event, which will be processed by the main executive to begin the dumping of a single image frame.
SetCameraRegsCmd	Take the set of I2C register addresses/data sent as part of the command, and send them to the OV6620 to update its internal registers.

EnableTrackingCmd	Start color blob tracking by publishing the EV_ENABLE_TRACKING event, which will be processed by the main executive to begin color tracking based on the current colorMap.
DisableTrackingCmd	Stop color blob tracking, which will cause the system to go back to the idle state.
SetColorMapCmd	Update the colorMap data structure with the new colorMap passed in by the user. This also updates the version of the colorMap maintained in EEPROM.

There are a few additional functions in UIMgr.c that support the processing of commands, but these are fairly self-explanatory.

5 Color Tracking Logic

5.1 Files Involved:

AVR mega8 – FrameMgr.c, CamInterface.c, CamInterfaceAsm.S, UIMgr.c, Executive.c

5.2 Description

The color-tracking capability is the primary functionality provided by the AVRcam. The system provides the user with the ability to track up to eight different objects of up to eight different colors, at the full 50 frames/sec provided by the OV6620.

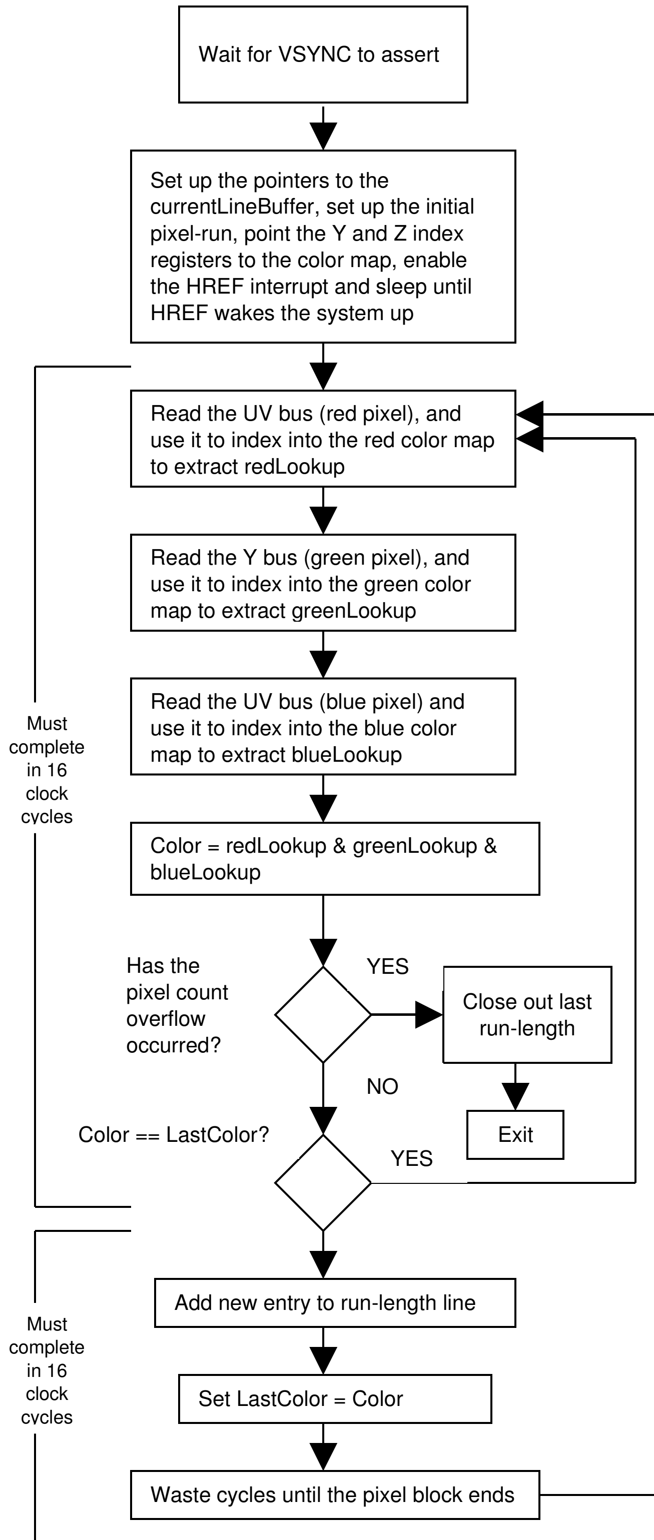
The color tracking feature begins when the system publishes the EV_ENABLE_TRACKING event after it receives the “ET\r” command from the user. The EV_ENABLE_TRACKING event is dispatched to the FrameMgr, where it sets a local state variable to ST_FrameMgr_trackingFrame and calls FrameMgr_acquireFrame. The FrameMgr_acquireFrame() function is called to start the acquisition and processing of an image frame based on the next time the VSYNC line asserts itself. This function first resets a few of the critical data structures used while processing the frame, and then calls the macro WAIT_FOR_VSYNC_HIGH(). This macro, defined in CamInterface.h, sits in a tight loop waiting for the VSYNC line to assert itself indicating that a frame is about to begin. Once this line asserts, the CamIntAsm_acquireTrackingLine() function is called, passing in a pointer to a buffer to be used for storing a line of run-length encoded pixels, as well as a pointer to the colorMap.

The CamIntAsm_acquireTrackingLine() routine is written in assembly due to the fact that it needs to execute according to a very specific timing profile. It must complete the processing of each incoming pixel in a fixed number of clock cycles to ensure that it is ready to process the next pixel. There is no time between pixels to read the PCLK signal (normally used to indicate valid data on the pixel data busses), so executing in lock-step

with the clock source for the image sensor is critical. This is why the OV6620 and the mega8 share the same clock source. In addition, there is no time between pixels to determine if a sufficient number of pixels have been received to indicate the end of an image line. Thus, the PCLK signal from the OV6620 is connected to the TIMER1 hardware timer/counter, which is pre-loaded to generate an interrupt after a complete line of pixels has been received (176 in total).

Ok…on to the code. The first thing CamIntAsm_acquireTrackingLine() does is to check the state of the "T" flag. This is a bit obscure, and is strictly not needed as a first step any longer. The "T" flag is a (typically unused) bit in the mega8's SREG register that was originally being used as a flag in the system to indicate any time a serial byte was received during the time-critical portions of the acquireTrackingLine() routine. If a serial byte was received, the associated serial-received interrupt would cause a hiccup in the processing and would thus trash the entire line of image data. However, this required the serial-receive ISR to be written in assembly language, due to the necessary setting of the "T" flag (GCC was resetting this bit otherwise), and was somewhat difficult to follow. Thus, the serial-receive ISR was moved over to C, and if serial data is received during a frame it immediately causes the frame-processing to end (by publishing the EV_PROCESS_FRAME_COMPLETE event to force the system to wait for the new frame…see FrameMgr_dispatchEvent() for more details).

The remaining processing can be best described in a flow chart, broken up into four distinct steps:


Step 1
Wait for VSYNC to assert
Step 2
Set up the pointers to the currentLineBuffer, set up the initial pixel-run, point the Y and Z index registers to the color map, enable the HREF interrupt and sleep until HREF wakes the system up
Step 3
Must complete in 16 clock cycles
Read the UV bus (red pixel), and use it to index into the red color map to extract redLookup
Read the Y bus (green pixel), and use it to index into the green color map to extract greenLookup
Read the UV bus (blue pixel) and use it to index into the blue color map to extract blueLookup
Color = redLookup & greenLookup & blueLookup
Has the pixel count overflow occurred?
YES
Close out last run-length
Exit
NO
Color == LastColor?
YES
Step 4
Must complete in 16 clock cycles
Add new entry to run-length line
Set LastColor = Color
Waste cycles until the pixel block ends

There are a few important details to point out here for each of the above steps:

Step 1-The system originally went to sleep waiting for VSYNC to assert. This has since been updated with a simple loop continually checking when VSYNC went high, as noted previously.

Step 2-This initialization is important, and must complete before the HREF line asserts indicating the pixel data for the line is about to start. This initialization clears out pixelCount, loads a value of 0x50 into pixelRunStartInitial so that it overflows after 176 pixels have been received (this is then loaded into the initial value of the TIMER1 hardware counter), sets up the X index pointer to point to the passed in buffer where run-length encoded color data will be stored, and loads the Z index pointer to point to the beginning of the colorMap data structure. Note: it is important that the colorMap data structure be located on a 256-byte boundary. This allows for fast indexing into the colorMap by simply loading the lower-byte of a sampled pixel into the lower byte, thus providing a direct index into the colorMap without any further manipulation. This is enforced by an optional linker-flag in the makefile, as well as an explicit declaration of the colorMap starting address at the top of CamInterface.c

Next, TIMER1 is enabled to start counting the external PCLK pulses, and interrupt after 176 pixels are counted. The interrupt associated with the assertion of the external HREF line is then enabled. This will cause an interrupt to occur when the HREF signal goes high, indicating that a line of pixel data is about to start. Once enabled, the mega8 goes to sleep, only to be woken up by the assertion of HREF. When HREF is asserted, it takes fourteen clock cycles to process the interrupt, which does nothing but simply return. This sleep-until-woken-up method is the only way to provide a guaranteed time delay between when the HREF signal asserts and when code begins processing. If the code just sat in a tight loop checking the status of the HREF line, there could be up to three clock cycles of uncertainty as to when the HREF line asserted, which is simply too much in a system when the remaining portion of the routine is critically timed.

Step 3-Once woken up by HREF, the associated interrupt is disabled since it is no longer needed. A few NOPs are added in to provide two clock cycles worth of delay. This brief delay is used to line up the rest of the processing that is about to occur to ensure that the sampling of the pixel busses is happening when valid data is present. Note: At no point is the PCLK line checked to determine if the data is valid or not (as would normally be done in a synchronously clocked system). It is assumed that since the OV6620 and the mega8 have the same clock source, the two run synchronously, and that it is sufficient to simply utilize the instruction cycle timing to determine when valid data is available on the pixel data busses. It should also be noted that the fourteen clock cycle delay, along with the NOPs, cause the processing to actually skip the first few pixels in each line.

This effectively reduces the line width slightly (from 176). However, the PCLK signal is feeding the TIMER1 hardware even while the wake-from-sleep ISR is running, and thus maintains an accurate pixel count.

After the NOPs, it is time to start sampling pixel data. First, a red pixel value is sampled into the lower byte of the Z index pointer, followed by a green pixel value being sampled into the Y lower byte of the Y index pointer. These two samples are for the same PCLK pulse. It is necessary to clear out the upper-nibble of the Y index pointer since the upper nibble overlaps with the I2C hardware bus, and is not associated with the pixel data. Once cleared, the Z index register plus the RED_MEM_OFFSET (0) is used as a pointer into the colorMap to extract the red value in the colorMap that maps to the sampled red pixel. The "color" variable is used to hold this initial value read from the colorMap. Once completed, a blue pixel is sampled from the pixel data bus and written directly into the low byte of the Z index register, since it is now available. This corresponds to the next PCLK cycle. Then, the Y index register plus the GREEN_MEM_OFFSET (16) is used as a pointer into the colorMap to extract the green value in the colorMap that maps to the sampled green pixel. The extracted value is written into the greenData variable. Finally, the Z index register is used again to index into colorMap, although this time it has a blue pixel in it, so the offset into colorMap is the Z index register plus the BLUE_MEM_OFFET (32). The extracted value is written into the blueData variable. A logical AND operation is then performed between the "color" variable (currently holding the extracted red value from the colorMap), the greenData, and the blueData. Once this AND operation is completed, the color variable will be left with either no bits set, indicating that the sampled red/green/blue triplet didn't map to any color in the colorMap, or with one bit set, indicating that the red/green/blue triplet did map to a color. The bit position that is set indicates which of the eight colors matches the triplet. Note: It is up to the user to ensure that the colorMap doesn't result in more than one bit set after a logical AND operation of a value in the red-section, a value in the green-section, and a value in the blue-section of the colorMap. The AVRcamVIEW PC software automatically verifies this when entering in a color map, and notifies the user if there is an error.

Before proceeding, the routine checks to see if the "T" flag is set in the mega8's status register. The "T" flag is set if an ISR was serviced during this loop (the only ISR we care about at this point is the TIMER1 overflow interrupt, or SIG_OVERFLW1 located at the end of CamInterfaceAsm.S, indicating that 176 pixels have been sampled and thus the line is complete…if the "T" flag is set, the _cleanupTrackingLine routine is called, which does a bit of housekeeping and returns).

It should be noted here that the above color-mapping takes 10 clock cycles, which effectively causes the next pixel block to not be sampled. Thus every other four-pixel block of pixels is processed. Note that this doesn't throw off the actual counting of pixels in the line, since this is performed by the TIMER1 hardware counter. But it does reduce the effective accuracy of each tracked object from two pixels to four pixels.

Back to the processing loop. If the color we just found is the same as the last color, the system loops back to start reading the next set of pixels, since we aren't starting a new run-length. Otherwise, if it is different, Step 4 is executed, where we need to create an entry in the run-length with the appropriate number of pixels and the new color. This activity will cause the next set of red/green/blue pixels to be passed over for processing since the mega8 is busy recording the run-length change information, but PCLK will continue to increment the TIMER1 counter since this is all done in hardware. Once the run-length updating is complete, the whole sampling of the next set of red/green/blue pixels starts again.

Eventually, the TIMER1 interrupt will overflow, triggering the SIG_OVERFLOW1 interrupt handler. This handler sets the fast-event FEV_ACQUIRE_LINE_COMPLETE to indicate to the executive that a complete line has been received and run-length encoded. This handler also sets the "T" flag to indicate, as mentioned above, that 176 pixels have been mapped into colors and run-length encoded, and its time to process them. After exiting the ISR, the "T" flag being set will cause a jump to the _cleanUp routine, which disables the TIMER1 interrupt from occurring (it will be re-enabled when we start the next line of processing), and finally returns back to the FrameMgr_acquireFrame() routine that initially called it.

After each image line is processed, a run-length encoded data structure (currentLineBuffer) is populated and is passed back to the FrameMgr_acquireFrame(). This data structure has the following form:

Color1	Run-length 1	Color2	Run-length 2	Color3	Run-length 3	○ ○ ○

Where each "color" has either one-bits set (indicating which of the eight colors is represented by the run-length) or no bits set (indicating the run-length didn't map to any color in the colorMap. The "run-length" is the number of consecutive pixels in the line which contain the color. This currentLineBuffer is simply a global array of bytes (declared in CamInterface.c), and thus can be easily processed by the FrameMgr.

After the FramgeMgr_acquireFrame() routine completes, the FrameMgr_dispatchEvent() routine returns back to the main Exec_run() processing loop. Recall that the completion of an image line generated a FEV_ACQUIRE_LINE_COMPLETE event, which will now be processed by the Exec_run() function. This function will clear the event from its fast-event bitmask and call FrameMgr_processLine() to process the currentLineBuffer data that was just collected.

The FrameMgr_processLine() routine can process image lines in both the "dumping-frame" state as well as the "tracking frame" state. Since we're in the "tracking frame" state, the _processLine() routine calls FrameMgr_findConnectedness() to take the currentLineBuffer data structure and see if the various colors encoded into this image line

overlap in position with any of the colors found in the previous line. It is important to note here that the only thing that makes a difference is the currentLineBuffer data structure and a data structure called trackedObjectTable declared at the top of FrameMgr.c. The trackedObjectTable data structure is an array of trackedObject_t structures. Each trackedObject_t structure contains the pertinent information for each connected region of color that matches a color in the colorMap, including color, the start and finish index of where the color was found in the previous currentLineBuffer, and the x-y coordinates of the upper left and lower right corners of a bounding box that bounds the color blob.

The _findConnectedness() routine loops through all of the run-lengths encoded into currentLineBuffer to determine if each colorful run-length is connected to an object in the trackedObjectTable. "Connected" is defined as any portion of a colorful run-length overlapping its x-coordinates with the lastLineXStart to lastLineXFinish range of any object in the trackedObjectTable. If so, it is considered connected, and the bounding box coordinates for the already-present object are updated in the trackedObjectTable. If a colorful run-length is found and it doesn't overlap an already-existing object, a new entry is added to the trackedObjectTable with the needed parameters of this new object. A maximum of eight objects can be tracked at any one time in the trackedObjectTable.

Once the _findConnectedness() routine completes, the total trackedLineCount index is incremented. If the trackedLineCount is equal to the total number of lines in a frame (144), then the EV_ACQUIRE_FRAME_COMPLETE event is published, indicating that the frame is complete. Otherwise, the FEV_PROCESS_LINE_COMPLETE event is published to indicate that the individual line processing has ended. This completes the processing in FrameMgr_processLine(), and returns back to Exec_run().

Since we just completed the processing of an image line, the next thing we need to do is to check and see if any data needs to be sent out of our UART before we go back to start sampling the next image line. The UIMgr_transmitPendingData() routine is called, which checks to see if there is any data pending to be sent out in the UartTxFifo. If there is, only a single byte will be transmitted by the hardware UART at 115.2 kbps. The total time it takes to send a single byte at 115.2 kbps is 1/115200 * 10-bits per byte (with start and stop bits included) = 86.8 uS, which is less than the time it takes to acquire and process a single image line. One byte will be sent out of the UART per processed image line, allowing for up to 144 bytes to be sent out per image frame in a very efficient manner. Since each object in the trackedObjectTable occupies 8-bytes, and there are up to eight tracked objects (plus a little extra overhead to frame each message), the complete trackedObjectTable can efficiently be sent out interleaved with the processing of each image line.

After the _transmitPendingData() routine has completed, its back to Exec_run(). Recall that if we aren't at the end of the image frame, the FEV_PROCESS_LINE_COMPLETE event will be waiting to be serviced if there are remaining image lines in the frame to be

acquired/processed. This event will cause the FrameMgr_acquireLine() routine to be called. This routine waits for the HREF line to go low (which is the normal state of the line if it isn't asserted), and then calls CamIntAsm_acquireTrackingLine(), which begins the acquisition of the next processing line in the image.

After all 144 image lines have been acquired and processed, the EV_ACQUIRE_FRAME_COMPLETE event is published. This event is processed by the Exec_run() routine, which calls FrameMgr_processFrame(). The FrameMgr_processFrame() routine is responsible for manually building up each tracking packet to be sent on to the UIMgr's Tx FIFO to eventually be sent out over the hardware UART while the next image frame is being acquired and processed. It loops through each object in the trackedObjectTable checking for the object's validity, and will only send tracking packets out for objects that are valid.

Once the tracking packet has been inserted into the UIMgr's transmit FIFO, the EV_PROCESS_FRAME_EVENT is published. This event is processed by Exec_run(), dispatching it to the FrameMgr where the FrameMgr_acquireFrame() routine is called. This completes the entire processing of each frame when in color tracking mode. This acquisition/processing sequence continues until the user sends the "DT\r" command to disable tracking.

6 Frame Dumping Logic

6.1 Files Involved:

AVR mega8 – FrameMgr.c, CamInterface.c, CamInterfaceAsm.S, UIMgr.c, Executive.c

6.2 Description

The Frame Dumping capability of the AVRcam allows a user to request the system to take a snapshot image, and return it to the user. The AVRcam performs this operation by acquiring two lines at a time per image frame, over the course of 77 image frames, sending the contents of each set of two lines over the hardware UART back to the user. This operation takes approximately four seconds due to the 115.2 kbps baud rate of the UART.

Frame dumping starts by the user requesting this operation by sending the "DF" command to the system. Reception of this command will cause the EV_DUMP_FRAME event to be published by the UIMgr. The Exec_run() routine will then dispatch this event to the FrameMgr. The FrameMgr needs to first reduce the frame rate of the OV6620 image sensor by sending an I2C write command to register 0x11 with a new value of 0x01. This will reduce the frame rate to one-half its normal maximum frame rate. This is necessary because it is not possible to sample and store each pixel at the maximum

frame rate. At the reduced frame rate, it takes eight clock cycles to sample and store two pixels, which is sufficient for being able to store all the pixels in each line in real-time.

After the frame rate is reduced, the FrameMgr changes states to the "dumping frame" state, and calls FrameMgr_acquireLine(). The _acquireLine() routine performs its processing based on the current state. It first clears out the currentLineBuffer and previousLineBuffer which will be used to store two individual image lines. Recall from Figure 3 that one image line contains green pixels, and one contains alternating red and blue pixels. The mega8 is always sampling two lines at a time, grabbing one pixel from the "green" lines, and one pixel from the "red/blue" lines, thus sampling two image lines for each assertion of HREF. One line of image data will be stored in currentLineBuffer (green data), and one line of image data will be stored in previousLineBufffer (red/blue data).

The _acquireLine() routine then waits for VSYNC to toggle, indicating that a new frame is about to begin. Once this occurs, it is necessary to wait for HREF to toggle. However, since we are sampling successive line-pairs in each frame, over the course of 77 frames, it is necessary to wait for the appropriate number of HREF toggles to indicate that the line-pair of interest is about to begin. Once the appropriate number of HREF toggles have occurred, the CamIntAsm_acquireDumpLine() routine is called, passing in pointers to both the currentLineBuffer and the previousLineBuffer where the pixel data from each pair of image lines will be stored.

The _acquireDumpLine() routine performs almost identical initialization processing as the _acquireTrackingLine() routine to setup the TIMER1 counter to count the PCLK transitions. The only difference is that it sets up the X index register to point to the currentLineBuffer, and the Y index register to point to the previousLineBuffer. This will facilitate efficient writes to these buffers. The next difference in processing doesn't occur until the _sampleDumpPixel label in the CamInterfaceAsm.S file.

The _sampleDumpPixel label marks the code that actually performs the sampling of the Y and UV data busses to grab the individual pixel data. This routine samples each pixel data bus and stores the samples to the X/Y indexed buffers. This processing must complete in eight clock cycles, as noted before. Once 176 pixels have been sampled, the TIMER1 overflow interrupt handler (SIG_OVERFLOW1) will be called, which publishes the FEV_ACQUIRE_LINE_COMPLETE event as well as setting the "T" flag in the mega8's status register. The setting of the "T" flag causes the pixel-sampling loop to exit, and for the code at _cleanUpDumpLine to execute. This disables the external clocking of the TIMER1 counter so that the PCLK doesn't feed it any longer, and then returns back to the FrameMgr_acquireLine() routine, which returns back to the Exec_run() routine.

Back in Exec_run(), the FEV_ACQUIRE_LINE_COMPLETE event is processed. This causes the FrameMgr_processLine() routine to be called. The _processLine() routine

operates based on the state of the Frame Mgr. In the “dump frame” state, it builds up the beginning of a frame-dump packet and sends it directly to the UART. This is done, instead of buffering it in the UIMgr’s Tx FIFO, because we want to send out this data immediately. Next, the routine loops through and combines the corresponding “i index” pixel sample from both the currentLineBuffer and previousLineBuffer into a single 8-bit value where the upper nibble contains the 4-bit (green) pixel sample from the currentLineBuffer, and the lower nibble contains the 4-bit (blue) pixel sample from the previousLineBuffer. This packed sample is then sent out the UART. Next, the loop reads the “i+1 index” pixel sample values out of the currentLineBuffer and previousLineBuffer, and performs a similar operation. But this time, it flips the samples around so that the upper nibble contains a 4-bit (red) pixel sample from previousLineBuffer, and the lower nibble contains the 4-bit (green) sample from currentLineBuffer. This way, the frame dump data packet sent to the user contains properly formatted Bayer color data, as referenced on page 28 of [1].

Once all the bytes are sent out for both the currentLineBuffer and the previousLineBuffer, the lineCount is incremented. Since two image lines are sent out per iteration of FrameMgr_processLine(), lineCount only counts up to 72 instead of the expected 144 (which is the actual number of lines in an image). If lineCount hasn’t reached 72 yet, the event FEV_PROCESS_LINE_COMPLETE is published to kick off the acquisition of the next set of sequential dump lines after the next image frame starts. If lineCount has reached 72, the FrameMgr returns to the “idle” state, resets lineCount back to 0, and sends a command to the OV6620 to return to the full frame rate by setting the I2C register address of 0x11 with a value of 0x00. After this command is sent, the FrameMgr_processLine() routine returns back to Exec_run(), looping and waiting for a command from the user.

7 Summary

Hopefully, this write-up provides the reader with a better understanding of how the AVRcam works, as well as removing some of the mystique of how a real-world image processing system operations. The source code itself is reasonably well commented, and should be referenced to help clarify what exactly is going on as questions arise. If there are additional questions regarding the system, post them to:

http://www.jrobot.net/Forums.

Thanks again to John Sosoka for funding this write-up, and allowing me to share it with others. Good luck with the Pleo!