CamConnect
A USB Enabled, Embedded Internet Reference Design

1 Overview

By integrating the technologies of Mitsubishi's M30240 USB MCU and M64283 CMOS Image Sensor device with emWare's embedded Internet connectivity, this demo allows the user the ability to view continuously captured still images from a remote location with nothing more than a standard web browsing client.

The unit is powered by the M16C/240 Full Speed USB MCU running at 12MHz. The demo's Windows interface allows for easy connectivity to the demo board under a Windows98 platform. A Mitsubishi M64283FP CMOS Image Sensor captures 128 x 128, 8 bit grayscale images and passes them back up to the client software through the USB.

CamConnect boards

The Image capture device is also mounted on a motor driven, rotating platform which enables the user to remotely change the current view at any time through the web browser. Using emWare's Legacy connectivity software, the device is made viewable to the world through the Internet. The interface allows for not only control of the image capture, but also monitoring and setting of camera preferences (eg: exposure adjustment) and rotation of the camera.

2 System Components

Mitsubishi M16C USB MCU - M30240
- 16 bit MCU (M16C Series)
- Full Speed USB – 128 byte FIFO
- 12 MHz Operation
- 8 Bit A/D
- 2 DMAC
- 16 Bit Timers

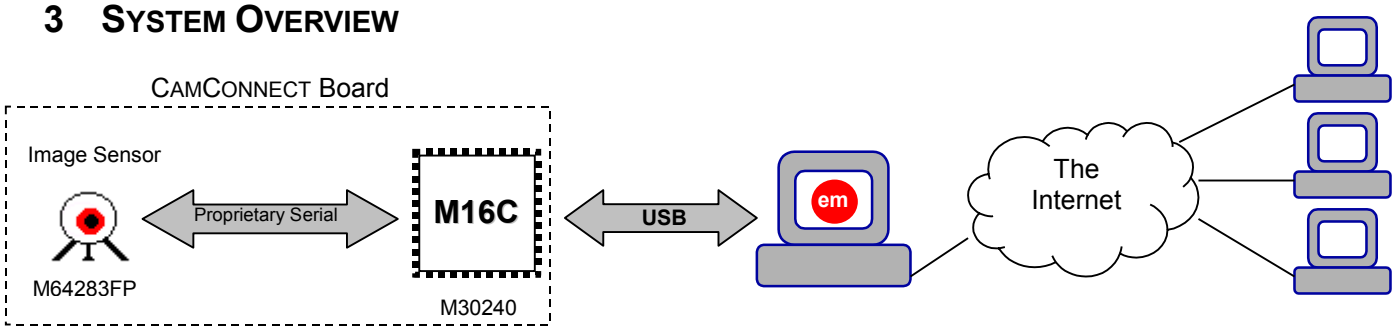
Mitsubishi CMOS Image Sensor - M64283FP
- 128 x 128 8 bit Monochrome
- On Chip Image Processing
  - Horizontal / Vertical Edge Enhancement
  - Horizontal / Vertical Edge Extraction
  - Invert Image

Win32 User Application – CamConnect.exe
- Controls Image Sensor Operations via USB
- Displays Captured Images in separate Window
- Manual & Automatic Exposure Adjustment
- Camera Motion Control
- Edge Extraction & Enhancement
- Remote Image Quality Adjustment

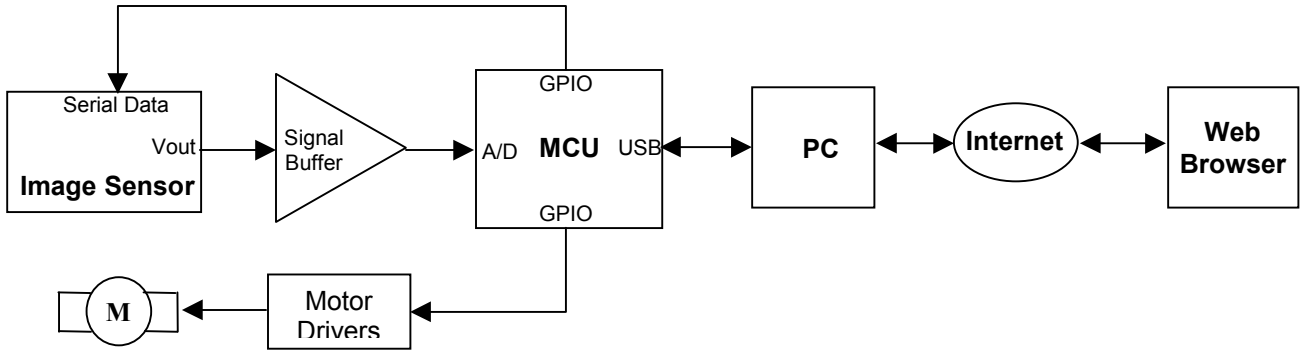
e^{TEK} Labs USB Bulk Driver
- Supports Control Transfers
- Supports 1 Bulk IN & 1 Bulk OUT Pipe

emWare's Internet Enabling Technology
- ActiveX Controls for Communicating with emGateway
- Java Web Applet
  - Requires no more than standard web browser
  - Provides user with ability to view captured images
  - Allows Control of Image Sensor Operations

3 System Overview



3.1 Block Diagram



3.2 General Description

- The Image sensor is controlled by the MCU using General Purpose IO pins
- The video signal from the Image sensor is buffered (level shifted and amplified) then fed into an A/D input on the MCU
- General Purpose IO pins are used to drive unipolar stepper motors
- The on-chip USB Serial Interface Engine and Transceiver is used to communicate to a Windows PC.
- The PC runs software that allows the images and functionality of the camera to be accessed via Java Applet running in a web browser.

4 USB COMMUNICATIONS

Communications between the Windows application and the MCU takes place over two defined USB transfer types: Control and Bulk. Control transfers are used for enumeration of the USB device as well as sending *Vendor Requests* that instruct the MCU which operation to perform.

4.1 Device Enumeration

What is USB Enumeration?
Enumeration is the process by which a USB device is attached to a system and is assigned a specific numerical address that will be used to access that particular device. It is also the time at which the USB host controller queries the device in order to decide what type of device it is in order to attempt to assign an appropriate driver for it.

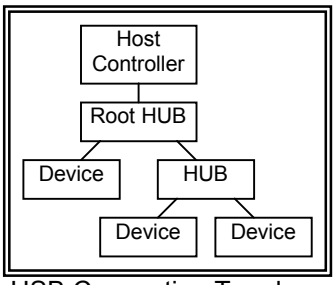
This process is a fundamental step for every USB device, fore without it, the device would never be able to be used by the OS.

What does enumeration look like?
I believe the easiest way to explain the USB enumeration process is to show it happening. The CamConnect demo firmware, for obvious reasons, contains code that will allow it to enumerate on any USB system. By using traces taken using a CATC USB Analyzer and snippets of code take from the CamConnect firmware source code, you should be able to follow and understand the enumeration process.

The following will be discussed:
- Plugging in the device
- Start of Frame
- Initial Communication
- Set Address
- Get Descriptor

4.1.1 Plugging in the device

All USB devices are plugged into a hub of some sort. When this is done, the hub detects whether the device is a full speed or low speed device. This is signified by the device pulling the D+ line to a 3.3v volt supply through a 1.5k pull-up for a full speed device, or the D- line for a low speed device.



USB Connection Topology

Once the hub has detected the connection of that new device, it will start passing Start Of Frame (SOF) packets produced by the host down to the device at 1ms intervals. The host controller will also start issuing setup packets to the device in order to enumerate the new device.

When a device is initially plugged in, it always uses the default device address 0 for communication. During the enumeration process, the host controller will assign a new numerical address for that device to use. Communication for the enumeration process always uses endpoint 0 on the device. These are considered to be Control Transfers. All USB Control transfers must use that device's endpoint 0.

After the host receives all the descriptors for the device, the OS will attempt to find an appropriate device driver to be associated with that new device.

4.1.2 Start of Frame

The following is an example of a SOF packet sent by the host at 1ms intervals. This SOF packet is sent to every device on the bus so that they may all be synched up. Every USB packet sent over the bus begins with a Sync pattern in order to allow all devices to synchronize their transceivers. A CRC calculation for that packet is also included for most packet types. The M16C USB hardware automatically takes care of these details.

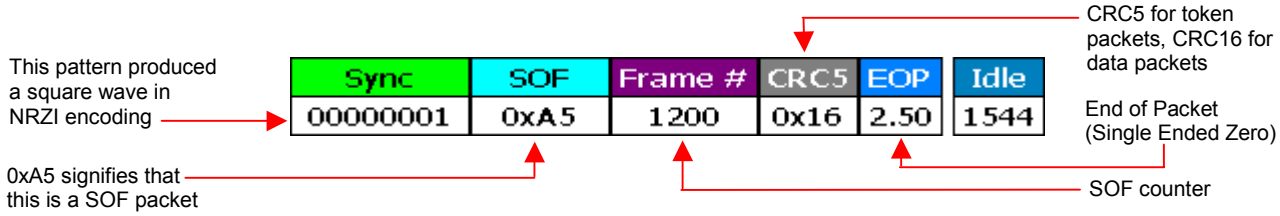


Figure 1. Start of Frame (SOF) Packet

4.1.3 Initial Communication

What first happens on the USB lines can be somewhat confusing to someone new to USB. Depending on how the USB host controller was implemented, you might see bus traffic that you would normally assume to be incorrect, but in actuality, it is a standard and necessary process for a USB host controller.

So what is this mystery? More or less, the USB Host Controller first asks the device before it does anything else for its Device Descriptor. The interesting part is that the device descriptor is 18-bytes long, but the host could care less about that and may only want the first 8 bytes of it. After it receives those, it will not even ask for the rest of the data from the device. Furthermore, the host will issue a reset for that line after which it will then start to send commands for USB enumeration.

The reason the Host does this is because the Device Descriptor contains the maximum payload size that is allowed for an endpoint 0 Control Transfer. This value is contained in the 8th byte of the Device Descriptor that we must send back to the Host. So the host first queries the device with a GetDescriptor command in order to just get this one piece of information. Once the host has determined that number, it resets the USB lines and starts the enumeration process.

The packet traces below are showing that a setup token was the first thing given to our device on EP0 that instructed to return the Device Descriptor.

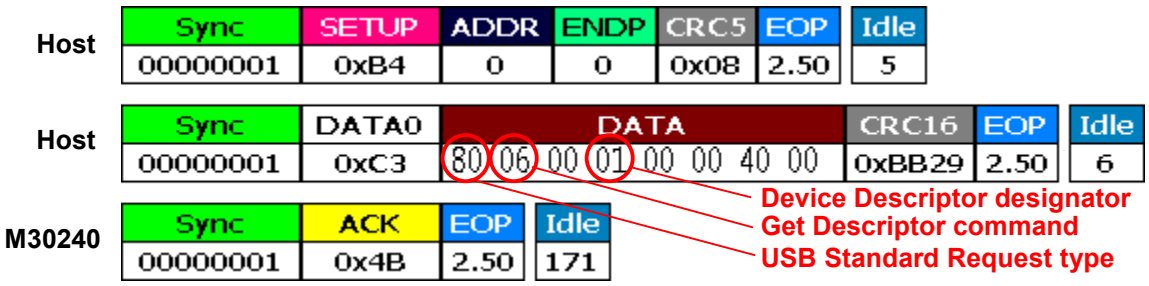


Figure 2. Initial Get Descriptor Request from Host

Below you will see that our device returned the first 8 bytes of the 18-byte descriptor, but instead of the host issuing us another IN packet so that we could transfer another 8-byte data packet, it issues an OUT packet followed by a NULL data packet. We ACK this transition, then the host controller resets the USB lines.

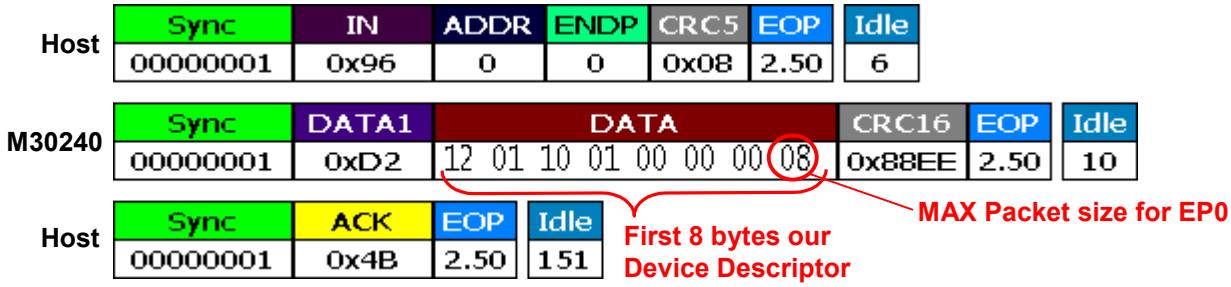


Figure 3. Our Device Answering the Initial Get Descriptor Request

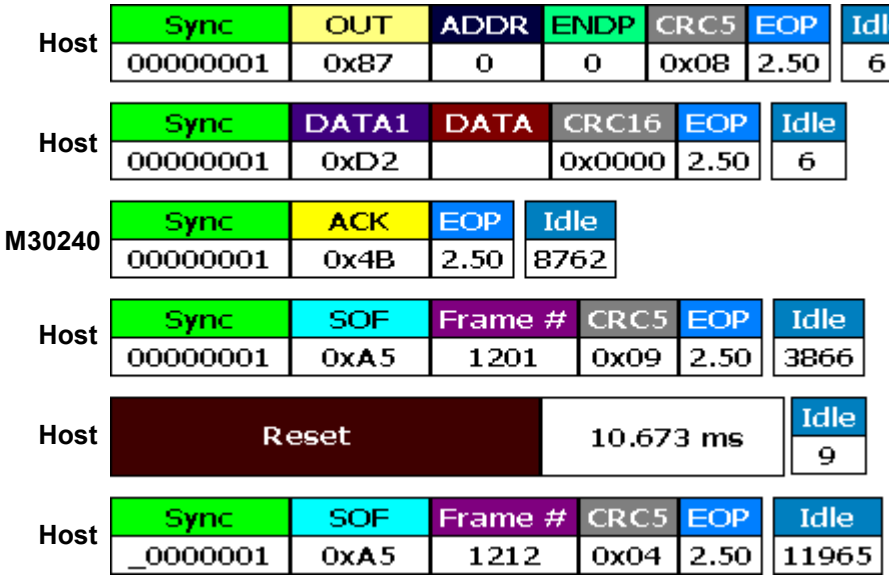


Figure 4. Host Resetting USB lines to begin enumeration

4.1.4 Set Address

For this enumeration process, the first command that was passed to the device was the Set Address command. As mentioned before, a new USB device on the bus temporarily uses a device address of 0 (zero) in order to provide a means of communication with the host. The host will then assign a specific numerical address for that device to use so that it will contain a unique identity on the USB bus.

Below you will see that a setup packet is sent to Device 0 , Endpoint 0, followed by 8 bytes of data that will be used to determine what type of setup packet is being sent, and what values need to be assigned. The M16C USB hardware contains a register that maintains the current device address. The register defaults to 0 after RESET, but can be written to at any time to change what address the USB hardware will respond to. The M16C USB hardware automatically sends an ACK back to the host saying that the data was received without error.

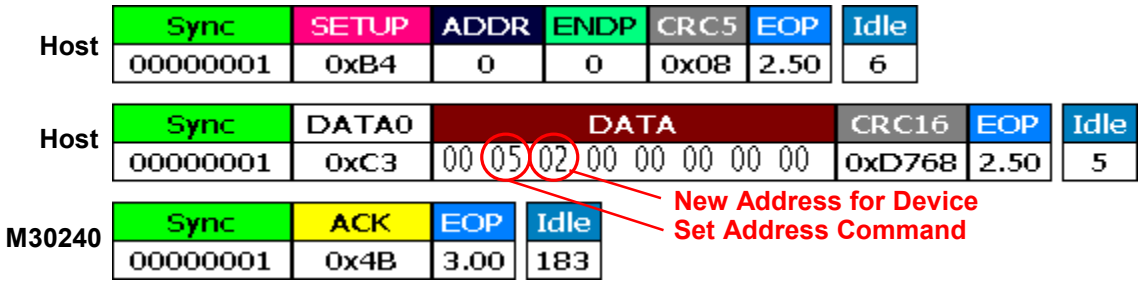


Figure 4. A Set Address command is sent to the device

Now lets take a look at how this packet is decoded in CamConnect's firmware.

The M16C's USB hardware will automatically accept and respond to device requests addressed to its current device address. Since the address register is set to 0 at the beginning of enumeration, our device will respond to all packet addressed to device 0.

Any time a USB packet is sent to a device address that matches our current device address, the M16C USB hardware will generate an interrupt (USBF– USB Function Interrupt). The interrupt handler can then query the USB registers to find out what caused the interrupt and respond accordingly.

Below is the USB Interrupt Service Routine for a USBF Interrupt.

```
void USB_Int_Handler() {

        /* Save and clear the current EP interrupts */
        USB_IntReg1 = usbis1;             /* USB Interrupt Status Register 1 */
        USB_IntReg2 = usbis2;             /* USB Interrupt Status Register 2 */
        /* Write this value back in order to clear those interrupts */
        usbis1 = USB_IntReg1;
        usbis2 = USB_IntReg2;

        /* We will use the mirrored variables for checking endpoint interrupts */

        /* == Check for EP0 Interrupt Status Flag ==*/
        if( USB_IntReg1 & 01) {
              ParseEP0Packet();           /* Service EP0 Request */
              USB_IntReg1 &= 0xFE;        /* Clear USBINT0 bit in mirror */
        }
        . . . .
        . . . .
}
```

As you can see, we first save the Interrupt status registers, and then write the values back in order to clear any bits that were set. We then use this information to decide what endpoint caused the interrupt. Since this is a setup packet, it would be Endpoint 0. We then call the appropriate function to handle this, ParseEP0Packet(), which is shown below.

```
void ParseEP0Packet() {

        if(ep0csr0 !=1 )     /* Check Out Packet Ready flag for EP0 set */
        return;              /* FIFO not ready to be read to return */

        /* Read the out 8 byte header from EP0 FIFO*/
        EP0_Header.bmRequestType = ep0;
        EP0_Header.bRequest      = ep0;
        EP0_Header.wValueLow     = ep0;
        EP0_Header.wValueHigh    = ep0;
        EP0_Header.wIndexLow     = ep0;
        EP0_Header.wIndexHigh    = ep0;
        EP0_Header.wLengthLow    = ep0;
        EP0_Header.wLengthHigh   = ep0;

        /* Mask out all but request type ( 01100000 ) */
        tmp_byte = EP0_Header.bmRequestType;
        tmp_byte &= 0x60;

        switch( tmp_byte ) {

                case 0:       ProcessStandardReq();      // USB Chapter 9 stuff
                              break;
                case 0x20:    ProcessClassReq();         // Specific Class stuff
                              break;
                case 0x40:    ProcessVenderReq();        // Custom stuff
        }
}
```

Above you can see that the 8-byte data packet that was sent was read out of endpoint 0 into a variable structure. Note that every setup packet has the same 8-byte format in which data is sent. Once the data is read out, it can be analyzed. You can see that we look at bits 5 and 6 of the first byte, otherwise known as the bmRequestType in USB lingo, in order to determine which type of request is being made. All USB enumeration requests are made via Standard Requests.

The function ProcessStandardRequest() that is shown below is then called which will then further decode the packet data.

```
void ProcessStandardReq() {

        /* Determine what is being requested */
        switch( EP0_Header.bRequest ) {
                case 0:       CmdGetStatus();
                              break;
                case 1:       CmdClearFeature();
                              break;
                case 3:       CmdSetFeature();
                              break;
                case 5:       CmdSetAddress();
                              break;
                case 6:       CmdGetDescriptor();
                              break;
                case 7:       CmdSetDescriptor();
                              break;
                case 8:       CmdGetConfiguration();
                              break;
                case 9:       CmdSetConfiguration();
```

```
                        break;
                case 10:    CmdGetInterface();
                        break;
                case 11:    CmdSetInterface();
                        break;
                case 12:    CmdSynchFrame();
                        break;
                default:    ep0csr = 0x44; // Clear out pky ready with send stall
                        ep0csr2 = 1;    // Stall all subsequent transactions
                        asm("nop");
                        asm("nop");
        }
}
```

By using the second byte in the data packet sent by the USB Host, otherwise known as the bRequest, we can determine what type of setup command is being administered. All the possible USB standard requests as noted in the USB specification are listed above as well.

You will notice that from the packet diagram shown earlier, that it is a 5, which correlated to being a Set Address command. We then call the appropriate function to service this request which will be CmdSetAddress() as seen below.

```
void CmdSetAddress() {

        /* Load our new device address */
        usba = EP0_Header.wValueLow;

        /* Set DATA_END and OUT_PKT_RDY bit for EP0 */
        ep0csr = 0x48;
}
```

At this point, we have now determined what type of setup packet request has been administered to us. Seeing that it is a Set Address command, all we need to do is instruct our USB hardware to start accepting data for a new device address and respond back to the host that we understood the request and completed the task.

The variable usba is actually a symbolic link to the M16C's USB device address register. By writing a new value to that register, the USB hardware will automatically start responding only to that new device address. You can see that this value is passed from the host in the lower byte of the wValue word (the 3rd byte in the 8-byte data stream). Cross-referencing that information to the setup packet in Figure 4, we can see that our new device address will be 2.

Finally we set the DATA_END and OUT_PKT_RDY bits for endpoint 0 in the M16C's USB registers which will cause the M16C USB hardware to send back a 0-length data packet (also call a NULL Data Packet) back to the host. This will signify that we have satisfied its Change Address request. This can be seen below.

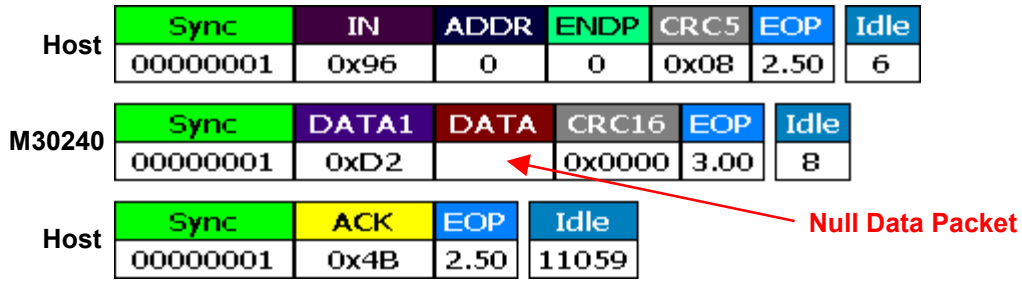


Figure 5. Device Acknowledgement of Change Address Request

The host sends an IN Packet Request to us (as we are still sitting at address 0) in order to receive conformation. As you can see, we respond correctly with a NULL data packet. The reason for the NULL data packet is that according to the USB spec, a device may NAK an IN token from the host as long as it wants. The host will

simply keep sending IN tokens to that device until an answer is received. In this case, the IN token is being used by the host to say, “Are you ready to start accepting data at your new address?” The M16C USB hardware will automatically send NAK packets back for us until our firmware has completed that task and we have set the appropriate USB registers. So, the Null data packet is like saying, “Yes, now I am ready, you may continue.”

An ACK is sent back from the host signifying that the response was received correctly.

4.1.5 Get Descriptor

The rest of the enumeration process is similar to this. It is like a system of questions and commands from the USB Host Controller for the newly attached device to follow. When the host is satisfied that it has enough information to search for an appropriate driver, it will stop sending setup packets for Standard Requests. At this point, the device is considered enumerated.

The GetDescription is another important setup command in the enumeration process. Unlike the first GetDescriptor issued by the host at the very beginning, this time we are expected to pass the entire descriptor back to the host. The setup packets are shown below.

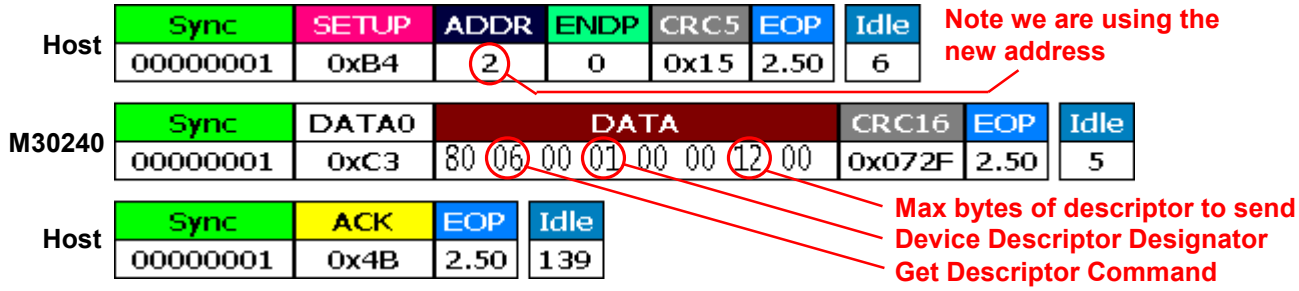


Figure 6. Get Device Descriptor Command from Host

Our firmware would then follow these steps…

```
void USB_Int_Handler()      // USB Interrupt Sub Routine
   ↳ void ParseEP0Packet()      // EP0 Control packet parsing function
        ↳ void ProcessStandardReq()  // USB Standard Request
             ↳ void CmdGetDescriptor()    // Service GetDescriptor command
```

In the CmdGetDescriptor() routine, we will then break up our 18-byte descriptor response into 8-byte data packets. Only when the host issues an IN token to us may we then transfer the data back up to the host in our firmware routine. We simply wait for the next IN packet to the host, fill up the endpoint 0 FIFO, then set the IN_PKT_RDY bit for EP0 in one of the M16C’s USB registers, and the hardware then transfer it up to the host.

Shown below are the bus traces doing just that.

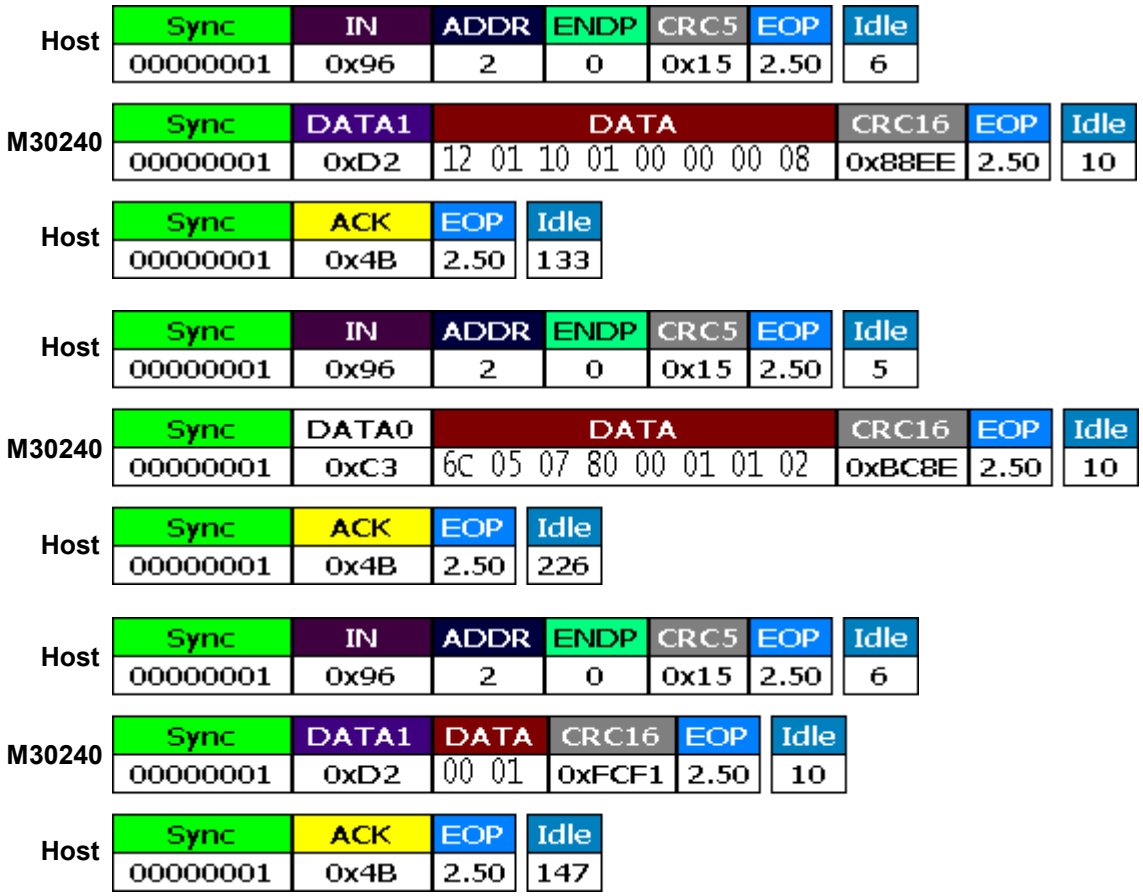


Figure 7. The M30240 Device Sending the 18-byte Device Descriptor 8 bytes at a time

This same procedure of breaking up the packets into 8-byte or less payloads will be done for various other USB commands such as getting the Configuration Descriptor, String Descriptor, Interface Descriptors and so on. This is because we told the USB Host Controller at the beginning that our MAX Packet size would be 8 bytes.

4.2 Sending Custom USB Command Vendor Requests

Vendor Requests are USB defined packets that enable the designers of the system to send miscellaneous information over USB to their device. This information resides in Setup Packets and could be any proprietary information the designer wants. The table below shows the proprietary commands used in this demo in Vendor Request format.

Command	bRequest	wValue	wIndex	Description
Get Image	0x03	-	-	Begin capturing a new image and fulfilling bulk IN token requests
Set Exposure	0x04	Exposure value	-	Load a new exposure time for the Image sensor
Set AR Register	0x05	New value	Register	Load a new value into any of the Image Sensor's registers
Set LEDs	0x69	LED pattern	-	Turns On/Off the LEDS
Move Motor	0x08	Pitch or Yaw	-	Change position of motor

Figure 8 Table of Camera Commands

4.3 Scheduling USB Commands

The MCU Software for this demo retrieves commands via vendor requests but only stores this information while in the USB Interrupt sub routine. Because USB interrupts with can come at any time, we want to make sure that the current operation can finish before intervening with the Image Sensor. To do this, we set an appropriate software flag. Commands are only executed in the main loop of the program that uses these flags to determine which operation should be performed next.
If for example a command is sent to change the exposure time on the Image Sensor while an image is currently being read out, the subroutine ProcessVendorReq() will first copy the new value for the exposure time into the mirrored Image Sensor Registers structure, and then set the variable “change_cis_regs” so that the main loop will know to change the registers after the image has be read out and before the next one has begun to be read.

4.4 Transferring Images

This demo uses *BULK* packets in order to upload the image to the Windows application. The USB specification defines a BULK transfer as a packet of data with a maximum payload of 64 bytes and is ensures data integrity. For this demo, only one BULK IN (from device to host) pipe was used to transfer the image data back to the application. Since the image sensor captured a 128 by 128 monochrome image (16 Kbytes of data), 256 USB BULK transfers were needed in order to transfer the raw image data up the application.

A vendor request would first be sent requesting an image from the device. The device would then begin to transfer pixels out of the image sensor and place them into the Endpoint 1 IN FIFO. At this time, the Windows application would instruct the device driver associated with our USB device to retrieve BULK data. The USB Host controller would then begin to send IN tokens to our device and retrieving data until our device instructs the host that it has finished.

Upon receive of an IN Token, the USB hardware on the MCU will automatically start to transfer the data back up the host assuming that there is data already in the FIFO and the IN_PACKET_READY bit has been set in the USB EP1 register. A USB interrupt will occur for Endpoint 1 IN when the transaction has been completed signifying that we may fill up the FIFO again in order to prepare for the next IN token. The ISR for this operation evaluates how many packets of pixels have been sent out and if more are still needed.

According the USB specification, a device is to signify that they are finish by sending data by mean of using one of two method. The are

1) If the packet sent by the device was less then it’s MAX PACKET SIZE (our case 64 bytes)
2) If a NULL data packet is sent (a packet with no payload)

Because the size of our image is an even multiple of 64 (64 x 256 = 16,384) we need to send a NULL data packet in order to instruct the host that we are finish send data and we no longer need IN tokens issued to us.

5 MCU FIRMWARE IMPLEMENTATION

5.1 Main Routine

The function of the main routine shown in Figure 9 is to first set up all timers, peripherals and USB registers for operation. After this is done, the firmware sits in a “while(1)” loop and checks to see if a command has been sent by the user. Commands are sent when a user clicks a button or moves a slider on either the windows interface or Java web browser applet.

If a command is requested, variables are used as flags in order to signify that a particular command should be performed. The reason why all commands must be performed in the main loop is that you do not want to be starting new operations before other operations have finish. An example would be trying to change the exposure registers on the image sensor while it is currently capturing an image.

5.2 USB Interface

Whenever there is USB traffic for our device, the hardware will automatically accept the data and generate an interrupt to signal our firmware as seen in Figure 10. Because a single interrupt is generated for any USB transaction, we must first determine which endpoint caused the interrupt. If Endpoint 0 was the cause, the packet sent must have been a setup packet. The only other valid endpoint traffic for our application would be Endpoint 1 IN which is used to transfer images back.

By examining the EP0 setup packet, the firmware then determines if the request is a standard or vender specific request. In the case of a standard request, out device will return the appropriate data according to Chapter 9 of the USB specification. Standard requests are what are issued during enumeration. In the case of a vendor specific request, we would simply record what command the user wants to issue and set the appropriate flag to signal the main to perform that action when appropriate.

An Endpoint 1 IN interrupt will occur after a complete EP1 FIFO transfer. This is similar to a UART transmit interrupt firing when the transmit buffer becomes empty. The AD Interrupt routine keeps track of how many pixels have already been transferred out of the image sensor as seen in Figure 11, then a DMA channel automatically transfers that pixel value into the EP1 IN FIFO. The EP1 IN routine checks to see if the correct number of packets for a single image have been sent and either ends the data transfer to the host or signals the firmware to continue to transfer pixel data from the image sensor to the EP1 IN FIFO.

5.3 Image Sensor Control

The protocol of the image sensor device was not a standard protocol that the M16C’s hardware could support or emulate. Therefore, communication was achieved by using General Purpose I/O pins and manually clocking out the addresses and register values.

In order to change the registers in the image sensor, the software waited until the current image transfer was completed. If a USB vendor request was sent instructing to change the value of one or more of the registers, this would be done before capturing the next image.

5.4 Motion Control

Unipolar stepper motors were used to change the position of the camera. The driver circuit consists of GPIO ports from the MCU and darlington transistors. If a command is issued to change the position, the software waits till the current image has been captured, then the camera is moved.

By tying each coil end of the unipolar motor to a IO port on the MCU, the motor is stepped by energizing the individual windings in a specific pattern. A simple delay is used in between steps in order to create more torque. That pattern is reversed in order to make the motor turn in the opposite direction. By leaving one of the coil windings charged after the motion has finished creates a break effect that holds the camera still.

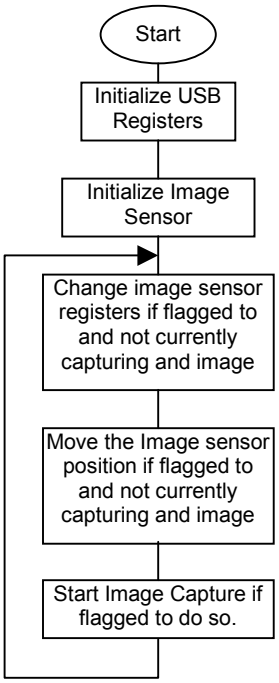


Figure 9 - Main Routine

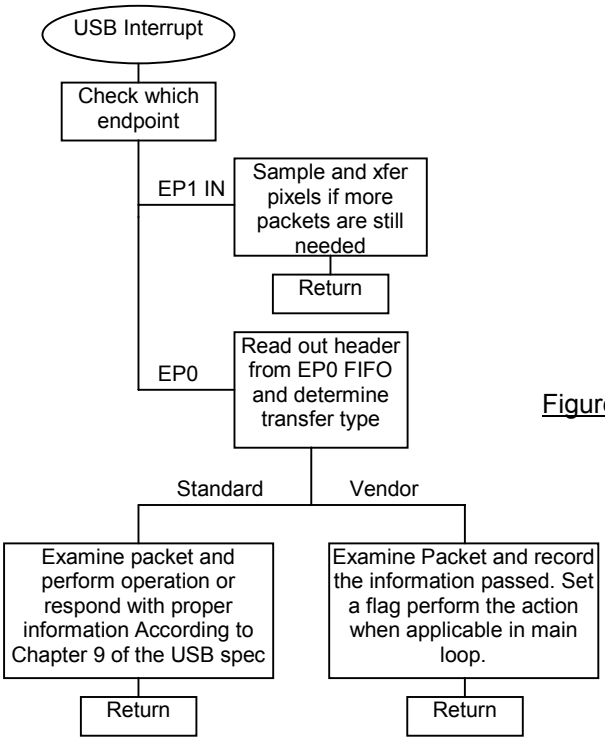


Figure 10 - USB Interrupt Routine

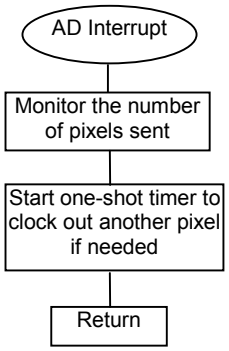


Figure 11 - A to D Interrupt Routine

6 CAMERA HARDWARE IMPLEMENTATION

The schematics for the both CamConnect boards which include the main CPU board and the attached image sensor board as well as the motor driver circuit can be found at the end of this document.

6.1 Main MCU Board

This board contained a socket that fit both the emulation pod connector as well as a UVEPROM version of the M30240EC. The MCU operates off a 12 MHz crystal and brings the signals that are needed for the image sensor interface out to a 16 pin header. The GPIO signals needed for the stepper motor control are brought out to an header as well.

There are 8 LEDs on the board that were used for debug purposes during development. Note that there is a specific LED driver port on the M30240EC that allows a direct connection to the LEDs with any type of LED driver.

Peripherals and ports used to interface to the image sensor board were:

- Timer Output – Create a set width clock pulse.
- AD Trigger Input – Trigger the AD to start a conversion on the video signal for the Image Sensor.
- AD Input – Sample the analog video signal.
- I/O Ports – Controlling and sampling the control signals.
- DMAC – Automatically transfer the sampled pixel value into the USB FIFO.

6.2 CMOS Image Sensor Board

The board was created in order to mount the image sensor devices for development purposes. Perforated area was left in order to experiment with different video buffer circuits.

The M16C's onboard AtoD was used to sample the analog video signal that was produced by the image sensor and a timer output was used to create a uniform clock signal. The AD Trigger pin on the MCU was used to signal when the signal was ready to be sampled. A DMA channel that was triggered off the completion of the AD Sample operation was then used to transfer the pixel value from the AD register to the EP1 FIFO.

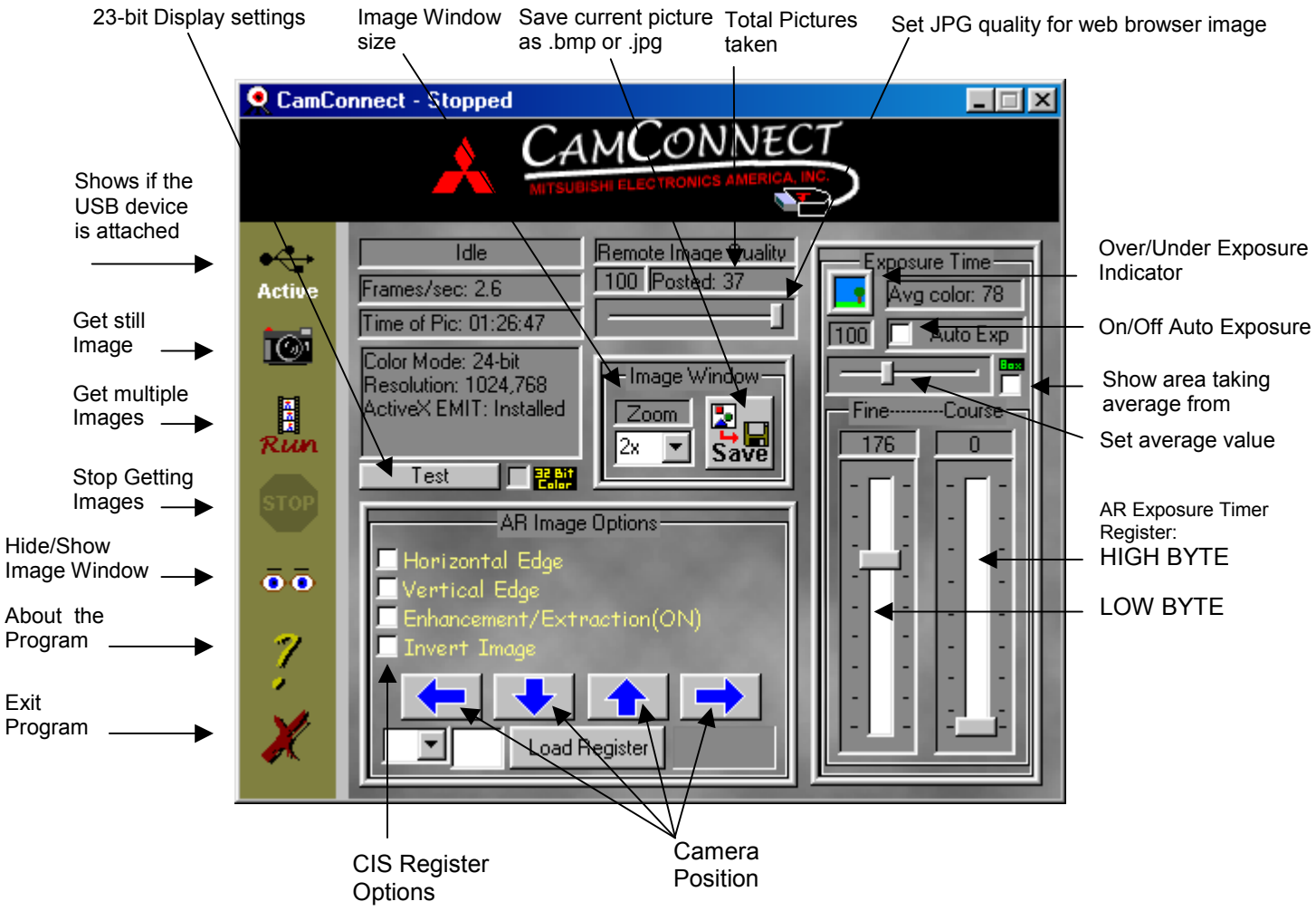
The analog video signal from the CMOS image sensor usually runs between 2 to 4 volts. It is possible to change the amplitude and offset voltage of this signal to some degree by changing the image sensor register values, but not enough to achieve a good 0-5 volt swing. Because of this, the addition of a level shift and amplifier circuit was added in order to achieve better resolution and image quality when sampling with the on-chip AD converter.

6.3 Stepper Motor Driver Circuit

A unipolar stepper motor was used because of its light weight and strong breaking ability. A simple darlington pair power transistor was used in order to supply the center tapped coils with 5 volts. No position detection circuitry was implemented in order to keep the design simple.

7 WINDOWS APPLICATION IMPLEMENTATION

7.1 Screen Shots



7.2 Application Operations

User Functionality
- Allow single or multiple image capturing
- Manual control of image sensor's 16-bit exposure time register. *Course* refers to the high order byte and *fine* refers to the low order
- Buttons become disabled when camera is not connected to PC
- Able to hide the image display window

Exposure Control
- Auto exposure takes the average brightness of pixels located within a defined area in the center of the view range
- An outline of the averaging area can be display
- The value that the auto exposure adjusts to can be changed. The exposure value will be altered after each captured image until the current exposure average is within ±5 of the set exposure average.
- The small picture of the tree indicates whether the auto exposure function is currently increasing or decreasing exposure time in order to adjust to the current brightness setting.

Remote Image Quality
- This setting is used in order to change the compression quality that the jpeg engine uses when creating a jpeg version of the captured image. A lower level of image quality will produce a smaller jpeg file. This setting may be used when a slow Internet connection is present and images cannot be transferred to the browser applet fast enough.

Statistics
- Frames/sec – Display the rate at which the current image was retrieved.
- Time of Pic – This is the time at which the image that is currently being displayed was captured. This information is also passed to the web browser applet.

Image Window
- Drawing – You can draw lines in black in this window by holding the left mouse button down.
- Saving – You can save the current image by right clicking anywhere in the image window.

Image Window Options
- Zoom – This simply stretches the image in the Image Window size for easier viewing. The resolution is still essentially 128 x 128 pixels.
- Save Image Button – This button is uses to save the current image that is being displayed in the image window. It can be saved as either a MS Bitmap (.bmp) or jpeg (.jpg) file. When saving as a jpeg, the image quality used is the same as the *Remote Image Quality* value.
- 32 Bit Color Option – This check box is automatically checked if the PC you are working on has the display color setting set to 32-bit color. Because of the way the images are drawn the screen, you must set the display settings of your PC to either 24 or 32 bit color. The test button can be used to verify that your settings are correct.

Image Sensor Options
- These options change the register values in the image sensor in order to utilize the built in feature of the M64283FP Image Sensor. The threshold settings are only general and could be fine tuned if desired in an actual application.

7.3 Windows Development

This Windows application was created using Microsoft's Developer Studio version 5. The first step in writing the application was to experiment with opening a handle to the USB device driver for our camera and passing data back and forth to the device.
Next, a user interface was created in order to allow the user to interact with device and do things such as request pictures and change register values located on the image sensor device. After receiving the raw 128 x128 pixel data from the device, it was painted into a separate dialog window for viewing. A JPEG compression was done on the image and saved to disk so that it may be transmitted to web browsers.

ActiveX controls provided by emWare (Xlink) were used to create a virtual device that could be connected to the emGateway software. As information was exchanged within the Windows app, it was also posted to the emGateway for any listening web browsers. Routines were also created to allow web browsers to interact with the device itself.

7.3.1 Development Libraries

The MFC libraries were used in order to create most of the controls used in the application. The API to emWare's emGateway was imported as an ActiveX component. The JPEG library was obtained from http://www.ijg.org/.

7.3.2 USB Driver Integration

A message handler was created that listened to system hardware insert and remove events. Every time a device was added or removed, the software would attempt to open a handle to the USB device driver that was

used for this demo. Depending on whether or not a handle could be obtained determined if the camera board was plugged in. The Windows application would then enable or disable the button controls depending on whether or not the demo was plugged in.

In order to create a communications *pipe* to the CamConnect board, a CreateFile() command is issued with the file name being specific to the USB driver loaded on the system. All following read/writes to that file stream would result in the appropriate USB traffic.

7.3.3 JPEG compression

An open source JPEG library was used in order to compress the raw image into JPEG form so that it may be displayed in a browser. The file was then saved to the hard drive under emGateway's HTML directory.
The current date and time was then recorded and passed to emGateway as a string. This would result in alerting any listening web browsers of this change which also in turn signaled them to retrieve a newer version of the JPG file from emGateway (or rather a new image to display).
Each image is deleted from the hard drive upon arrival of a new image from the USB driver.

7.3.4 Displaying the Image

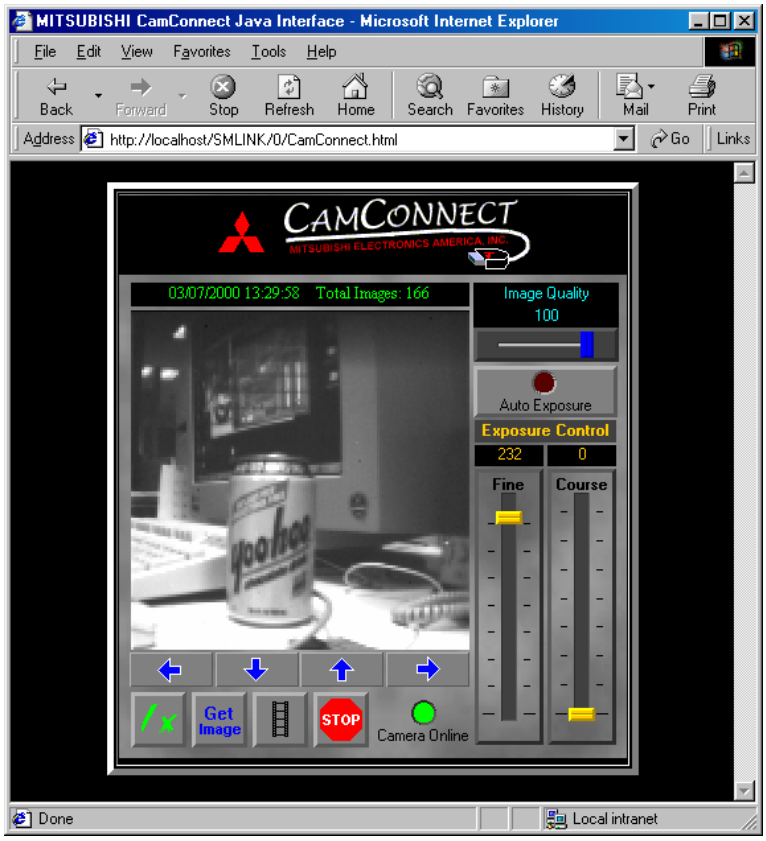
The Image was displayed by creating a CBitmap object and stretching it into a Device Context for the Image Window Dialog Window. The current *Zoom* setting determines the Image Window size and to what degree the image needs to be enlarged, if at all. When enlarging the image, a simple half-tone interpolation is used to create a larger image from the 128 by 128 pixels original.

The PC running this application must be capable of running at 24-bit or 32-bit color because of the way the CBitmap object is created and displayed. The application uses the registry to find out what the current color resolution setting is to determine if the application should run in 32-bit color mode or 24-bit color mode. If current color setting is below 24-bit color, a message box will inform the user of the problem and will bring up the system's display settings if they choose to set them before running.

8 INTERNET ENABLING THE SYSTEM USING EMWARE

8.1 Screen Shot

On the right is a screen shot of the Java Applet running in a web browser. Note the similar looking controls to the Windows application.



8.2 Overview

This demonstration uses emWare's Active X components called Xlink. Essentially, the Windows application would make itself appear as a virtual emWare device and connect itself to the emGateway via software. By doing this, the need for a USB Device Link Module (DLM) could be avoided. At the time this demonstration was created, there was no such DLM for emGateway available.

The ActiveX components allowed for transmission and reception of variables used to pass information back and forth between the Java applets and the CamConnect software. The CamConnect Windows software would interpret the information coming from the Java applets and then inform the M16C firmware my means of USB communications.

8.3 Bringing up the interface

A standard web page browser such as Internet Explorer® or Netscape® may be used to bring up the Java interface seen above. The URL that is entered is a combination of the IP address of the emGateway server that the USB camera is plugged into and the device name assigned to the camera board. For this demo, it was assumed that only one camera would be plugged in at a time, but the emWare system will support multiple devices or cameras plugged into the same emGateway.

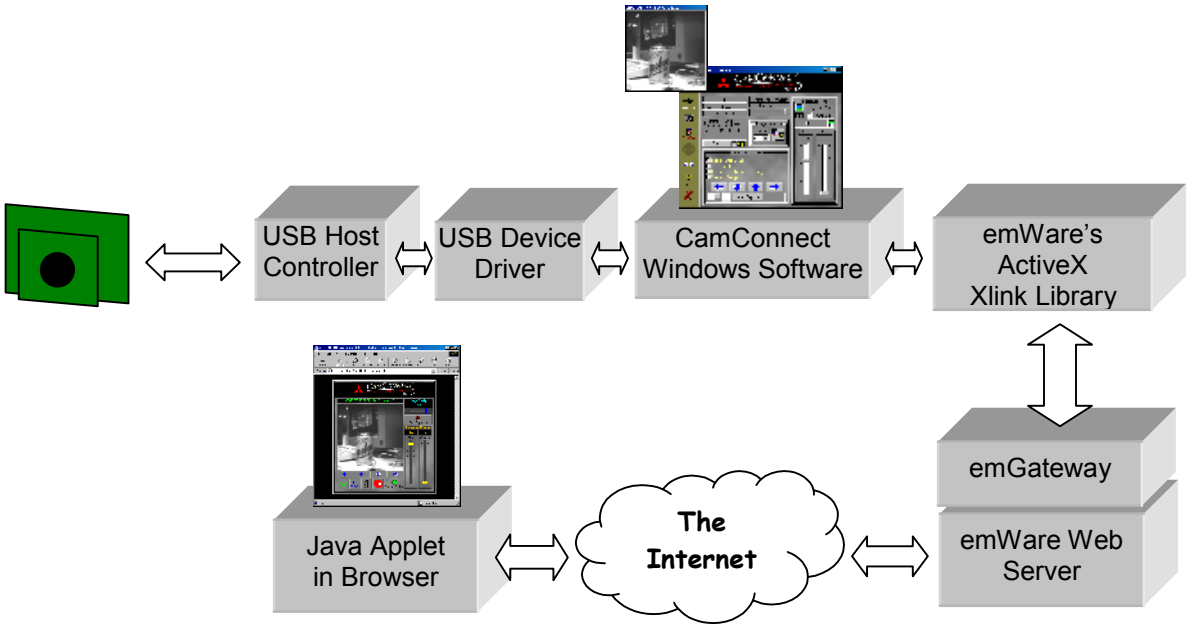
If for example the IP address of the emGateway machine was 10.1.1.5 and the device name assigned to the camera was CamConnect, then the URL that would allow you a connect to the device would be:

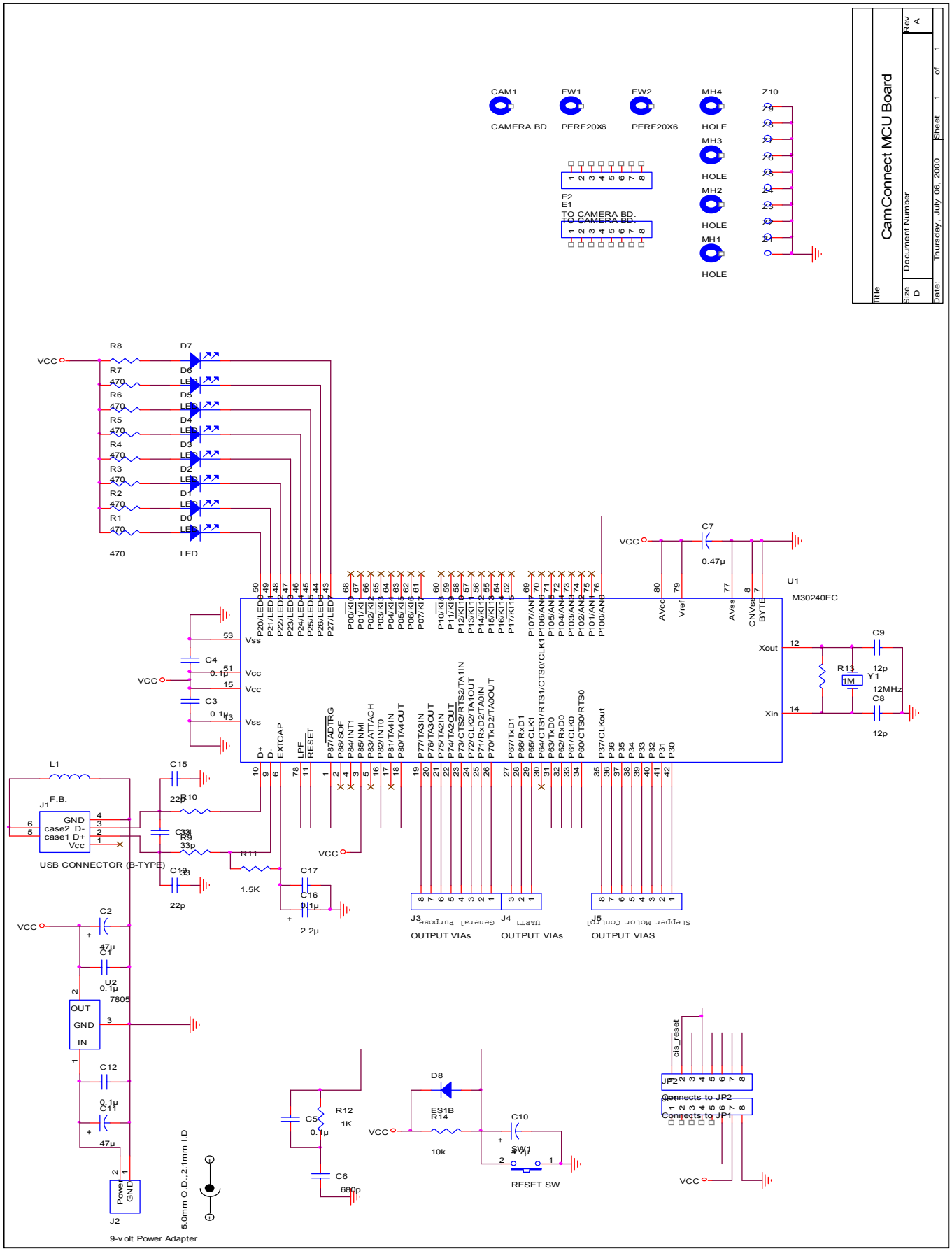
http://10.1.1.5/CamConnect

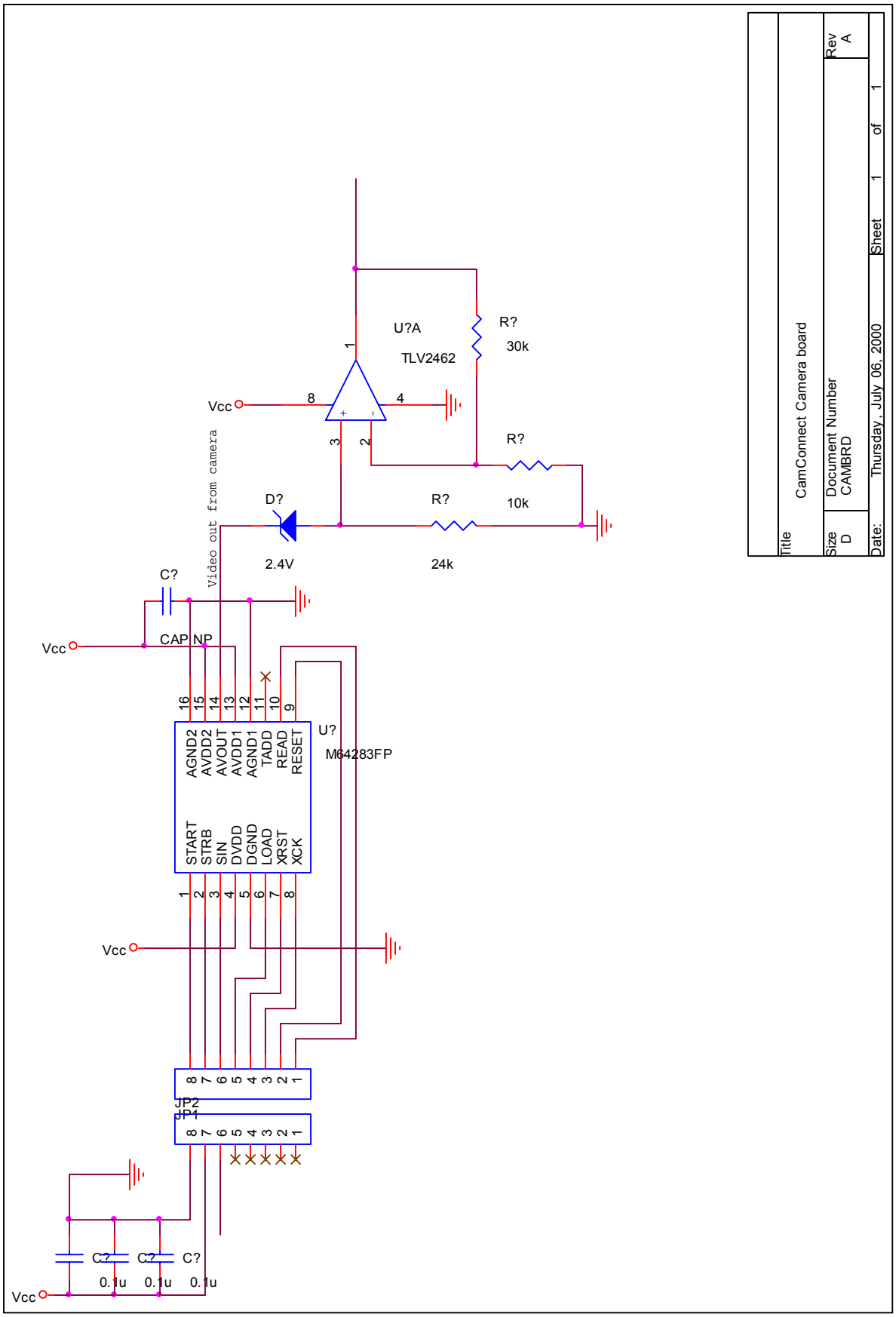
This would give you a list of interface files (.html) that you could choose in order to load the Java Applet to control your camera. The html file simply sets the background color and loads the Java class files. The Applet was created using Semantics Visual Café for Web development. A trial version of this software comes with Mitsubishi's ChipConnect EMIT SDK.

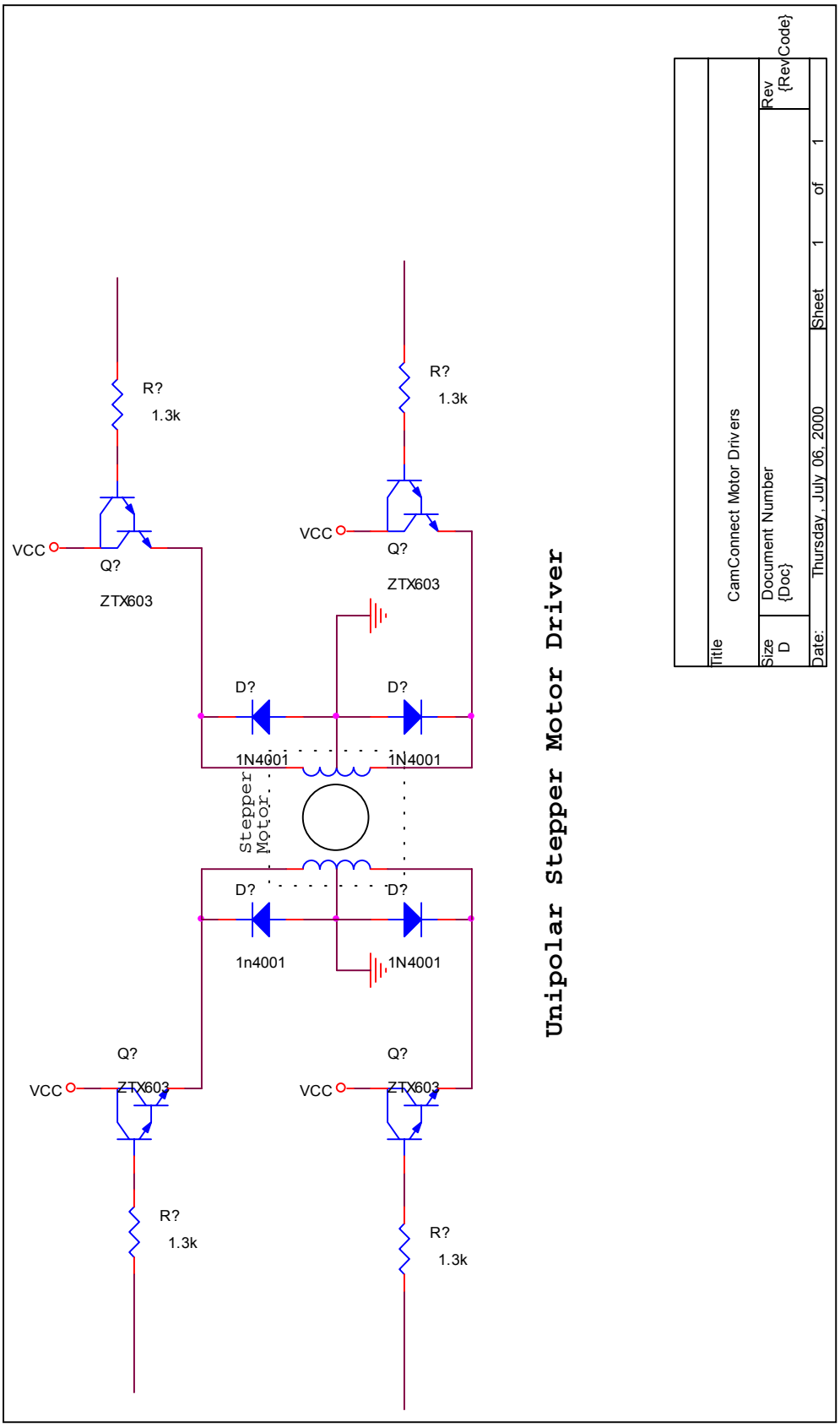
More than one browser may load the web interface in order to view images and control the camera at the same time. More implementation could have been done to allow a single user the ability to lock the controls and others to just view images.

From the diagram below, you can see that the all controls and images from the camera must pass through the Windows application. If an EMIT USB DLM was created instead of using the XLink library, the images and controls could be passed directly to the emGateway without the aid of the Windows application.




CamConnect MCU Board
Rev A
Size D
Document Number
Sheet 1 of 1
Date: Thursday, July 06, 2000
Title
CAM1
CAMERA BD.
FW1
PERF20X6
FW2
PERF20X6
MH4
HOLE
MH3
HOLE
MH2
HOLE
MH1
HOLE
Z10
E2
E1
TO CAMERA BD.
TO CAMERA BD.
R8
R7
R6
R5
R4
R3
R2
R1
470
D7
D6
D5
D4
D3
D2
D1
D0
LED
VCC
C7
0.47µ
U1
M30240EC
C4
0.1µ
C3
0.1µ
C9
12p
R13
1M
Y1
12MHz
C8
12p
Xout
Xin
L1
F.B.
J1
USB CONNECTOR (B-TYPE)
C15
22p
R10
C14
R9
33p
R11
1.5K
C13
33
22p
C17
C16
0.1µ
2.2µ
J3
OUTPUT VIAs
J4
OUTPUT VIAs
J5
OUTPUT VIAS
C2
47µ
C1
0.1µ
U2
7805
OUT
GND
IN
C12
0.1µ
C11
47µ
J2
9-volt Power Adapter
5.0mm O.D..2.1mm I.D
C5
0.1µ
R12
1K
C6
680p
D8
ES1B
R14
10k
C10
4.7µ
SW1
RESET SW
cts_reset
JP2
Connects to JP2
Connects to JP1


Title
CamConnect Camera board
Size
D
Document Number
CAMBRD
Rev
A
Date:
Thursday, July 06, 2000
Sheet
1
of
1
U?A
TLV2462
R?
30k
R?
10k
R?
24k
D?
2.4V
Vcc
Video out from camera
C?
CAP NP
U?
M64283FP
AGND2
AVDD2
AVOUT
AVDD1
AGND1
TADD
READ
RESET
START
STRB
SIN
DVDD
DGND
LOAD
XRST
XCK
JP2
JP1
C?
0.1u


Unipolar Stepper Motor Driver
Title
CamConnect Motor Drivers
Size
D
Document Number
{Doc}
Rev
{RevCode}
Date:
Thursday, July 06, 2000
Sheet
1
of
1
R?
1.3k
VCC
Q?
ZTX603
D?
1N4001
Stepper Motor
1n4001