

# ECP-II Project

Section ID: 35514 Embedded Controller Programming II: Embedded C

*Low-resolution color camera driver for  
the 8031-SDK*



Wolfgang Paulus

6606 Daylily Dr  
Carlsbad, CA 92009

mailto: wolf@paulus.com

Student ID:

## Table of Contents

<b>TABLE OF CONTENTS .....</b>	<b>2</b>
<b>1. ABSTRACT .....</b>	<b>3</b>
<b>2. HARDWARE DESIGN.....</b>	<b>3</b>
<b>2.1. BLOCK DIAGRAM.....</b>	<b>4</b>
<b>3. SOFTWARE DESIGN.....</b>	<b>4</b>
<b>3.1. SOFTWARE FLOW DIAGRAM .....</b>	<b>5</b>
<b>3.2. SERIAL COMMUNICATION .....</b>	<b>5</b>
3.2.1. SDK-PC TERMINAL COMMUNICATION.....	5
3.2.2. SDK-CAMERA COMMUNICATION.....	6
<b>3.3. CAMERA – DRIVER .....</b>	<b>6</b>
3.3.1. PROTOCOL.....	6
<b>3.4. IMAGE PROCESSING .....</b>	<b>8</b>
3.4.1. BAYER-DECODING .....	8
3.4.2. MISSING COLOR INTERPOLATION.....	10
3.4.3. BAYER DECODER, PSEUDO CODE: .....	11
<b>3.5. BITMAPS.....</b>	<b>12</b>
3.5.1. BITMAP FILE FORMAT .....	12
<b>3.6. UU-ENCODING .....</b>	<b>13</b>
3.6.1. PROTOCOL.....	13
3.6.2. IMPLEMENTATION .....	14
<b>3.7. ANALYSIS AND CONCLUSION .....</b>	<b>14</b>
3.7.1. TESTING AND SPECIAL RESULTS .....	14
3.7.2. CONCLUSION.....	14
<b>REFERENCES:.....</b>	<b>16</b>

## 1. Summary

This project demonstrates how to interface a low-resolution color camera, which is equipped with a serial port, with the 8032SDK.

A CMOS-based low-resolution (160 × 120 pixel) color camera's RS-232 serial interface is connected to the SDK's internal serial port. This connection is used to allow the SDK to control the camera and to download raw image data from the camera into the SDK's SRAM. The SDK's external serial port is connected to a PC/Terminal, providing a basic user interface, which allows for user controlled processing.

The low-cost digital camera does not provide any image processing and therefore, after a photo has been taken, about 20 Kbytes uncompressed, raw image sensor data, is downloaded into the SDK's SRAM memory. Since the camera's serial interface only support a single transfer speed of 57,600 baud, optimized assembler routines had to be implemented to allow the SDK to receive data and immediately transfer it into external memory at that speed. About 20 Kbytes are needed to store the raw image data for one image.

The readout happens in a horizontally shuffled Bayer colorization pattern and requires quite a bit of processing before a 24-bit per pixel MS-Windows compatible bitmap can be created. The full QSIF standardized bitmap (160 x 120 pixels with 24-bit color information), about 56 Kbytes, doesn't fit into the available external memory. Therefore, the image data is sent out through the external serial port while it is processed. Before the first pixel can be send however, a MS-Windows compatible BMP header is sent.

For the lack of availability of a better protocol, the receiving terminal has to log the receiving data, to eventually being able to display the bitmap. That however requires the SDK to send the data UUEncoded (standard for data interchange between systems with possibly different code sets, and to represent binary data as a text file).

Windows utilities like WinZip allow for convenient UUDecoding and image viewing in one single step.

## 2. Hardware Design

The Camera hardware consists of the following major components:

- STMicroelectronics VV6301 CMS Sensor
- TEMIC 51 X3702/B (Intel MSC51 compatible) Micro Controller
- Winbond 128 Kbytes CMOS SRAM
- 2 TI 97E1L1M High Speed Counter

The Camera's serial port is connected to the SDK's internal serial port, while the SDK's external UART connects to the PC / Terminal.

The Camera-SDK serial connection speed is set to 57,600 baud, the SDK-PC serial connection speed is set to 9,600 baud

## 2.1. Block Diagram

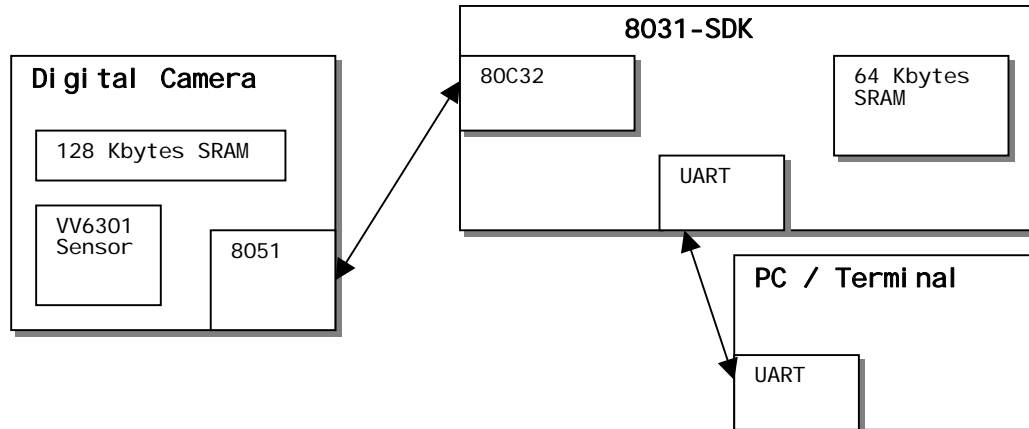


Figure 1 – Hardware Block Diagram

## 3. Software Design

To support re-use and better maintainability of the software, the following modularization was chosen:

- **Main.c** contains the loop that waits for and handles user inputs.
- **CamDriver.c** contains the camera specific functions, allowing control of the camera.
- **ImgProc.c** contains the implementation of the Bayer-decoding algorithm.
- **UARTex.c** contains replacements for STDIO lib function to allow I/O through the SDK's external UART
- **Serialin57.asm, Serialout57.asm**, high-speed serial driver for the SDK's internal serial port.

All modules come with \*.h files, containing function prototypes.

### 3.1. Software Flow Diagram

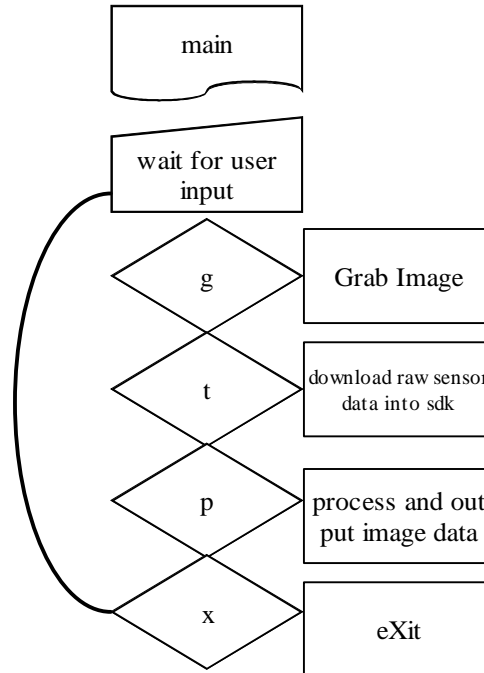


Figure 2- Software Flowchart

### 3.2. Serial Communication

This section covers the two serial connections used in this project. The SDK's external serial port is used to connect the SDK to a PC or Terminal, while the SDK's internal serial port connects the SDK to the camera.

#### 3.2.1. SDK-PC Terminal Communication

Serial communication between the SDK and the PC/Terminal happens through the SDK's external serial port at 9,600 to 38,400 baud. **UARTex.c** implements the following prototypes and replaces the STDIO library versions of *putchar(char)* and *\_getkey()*. The replacements implement I/O through the external serial port.

```

char putchar (char c);

char _getkey (void);

void print(char *s);

unsigned char ui();

void uuencode(unsigned char data *s, unsigned int n, unsigned char bpl );
  
```

*(Function prototypes in UARTex.h)*

### 3.2.2. SDK-Camera Communication

The serial communication between SDK and Camera happens through the SDK's internal serial port. Since the used camera only supports a single connection speed of 57,600 baud, serial port drivers supporting that speed had to be developed.

#### 3.2.2.1. Timer Calculation:

The following calculation determines that a timer in auto reload mode, needs to be set to 0xF0 to sample each incoming bit once.

$$11.0592\text{MHz} / (12 * 57,600\text{Hz}) = 16 \text{ and } 256 - 16 = 240 = \text{F0h}$$

*(Timer Calculation leads to reload value of 0xF0, used in **SERIAL\*.asm**)*

However, that leaves only 15 cycles to handle an incoming bit, putting it in the correct byte location and after receiving 8 bits storing the data byte in external memory.

Because of the tight timing requirements, the following two functions were implemented in assembler language. The registers for input and output variables were carefully chosen to allow the functions to be called from C functions.

```
void* sendserial(char data* ptr);

unsigned int readserial(unsigned int size, unsigned int xadr);
```

*(Function prototypes in **SERIAL57.h**)*

The implementation is located in **SERIALin57.asm** and **SERIALout57.asm**.

## 3.3. Camera – Driver

### 3.3.1. Protocol

All communications with the camera has to be initiated by the connected 80C32 controller, i.e., the controller is always the master and the camera always the slave.

All messages, both to and from the controller, are sent as packets starting with STX and terminating with ETX. [1]

**A command from the controller has the following format:**

**STX Command Data 1 Data 2 ... Data NETX**

**STX** Standard ASCII character (0x02).

**Command** A defined command byte, typically an **uppercase** character.

**Data n** Data for specific command. All commands have at least 1 byte of data (which is set zero if not used), some have more, the actually number is implicitly defined by the command byte.

**ETX** Standard ASCII character (0x03).

On receipt of a command from the controller, the camera always returns an acknowledgment. This will be either **ACK (0x06)** or **NAK (0x15)**. An ACK is returned if the command was successfully received and recognized (it does not indicate whether it was executed successfully, that is done by a response packet). A NAK indicates the command was not received successfully or was unrecognized. If no ACK or NAK is received in response to a command, the communication are assumed to be faulty or the camera disconnected.

**The response has the following format:**

***STX Response Data 1 Data 2 ... Data NETX***

**STX** Standard ASCII character (0x02).

**Response** A defined response byte, usually a **lowercase** character.

**Data n** Data for specific response. All responses have at least 1 byte of data (which is set zero if not used), some have more, the actually number is implicitly defined by the response byte.

**ETX** Standard ASCII character (0x03).

#### 3.3.1.1. Command: Reset Image Counter:

***STX CB\_RESET\_COUNTER (A) Index ETX***

The required image index must be in range 0 to MAX\_IMAGES or expect erratic results.

Response:

***STX RB\_RESET\_COUNTER (a) Error Code ETX***

Error Code is always zero. If the camera is busy the RB\_CAMERA\_BUSY response is returned with the appropriate busy code.

#### 3.3.1.2. Command: Grab an Image

***STX CB\_GRAB\_IMAGE (G) Control ETX***

The lower nibble of *Control* specifies the self-timer delay (0-15). A delay of zero means the grab is to be executed immediately. The upper nibble controls the response to bad lighting conditions. Bits 4-6 specify the number of retries if the exposure is bad, ie, 0, 16, 32 ...112. Bit 7 is reserved with the intention of implementing a “grab anyway” option if needed.

Response:

***STX RB\_GRAB\_IMAGE (g) Error Code ETX***

Error Code will always be set to zero. Use the grab result command to find out the status of the grab.

### 3.3.1.3. Command: Download Raw Image Sensor Data into the SDK

```
STX  CB_UPLOAD_IMAGE (U)  0  ETX
```

The argument is always zero.

Response:

```
STX  RB_UPLOAD_IMAGE (u)  N1 N2 N3 N4 D1 D2 ... Dn ETX
```

The complete image is returned as a stream of data bytes, including black lines, visible lines and status bytes. The total number of data bytes may be determined from the leading 4 bytes, N1 - N4, as follows:

$$N_{tot} = N1 \times (N2 + N3) + N4$$

**N1** is the number of columns in the image (164)

**N2** is the number of black lines (2)

**N3** is the number of visible lines (124)

**N4** is the number of status bytes (16)

$N_{tot}$  is the number of data bytes D1 - Dn only, it does not include the 6 leading header bytes or the final ETX. There is no explicit error reporting within the response message but the image status bytes may contain useful information.

## 3.4. Image Processing

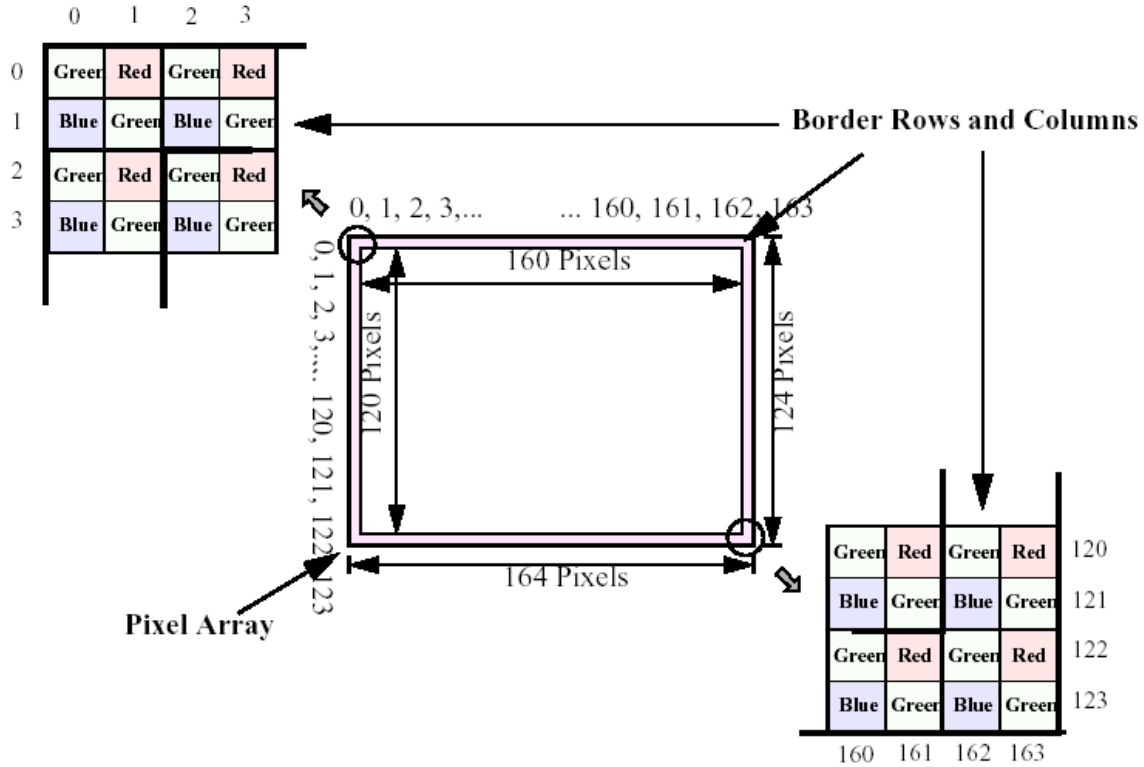
### 3.4.1. Bayer-Decoding

The *Bayer Pattern* is based on the premise that the human eye derives most of the luminance data from the green content of a scene; and it is the resolution of this luminance data that is perceived as the "resolution" of an image. Therefore, by ensuring that more of the pixels are "green", a higher resolution image can be created - compared with an alternating R-G-B color filter array with equal numbers of Red, Green and Blue pixels.



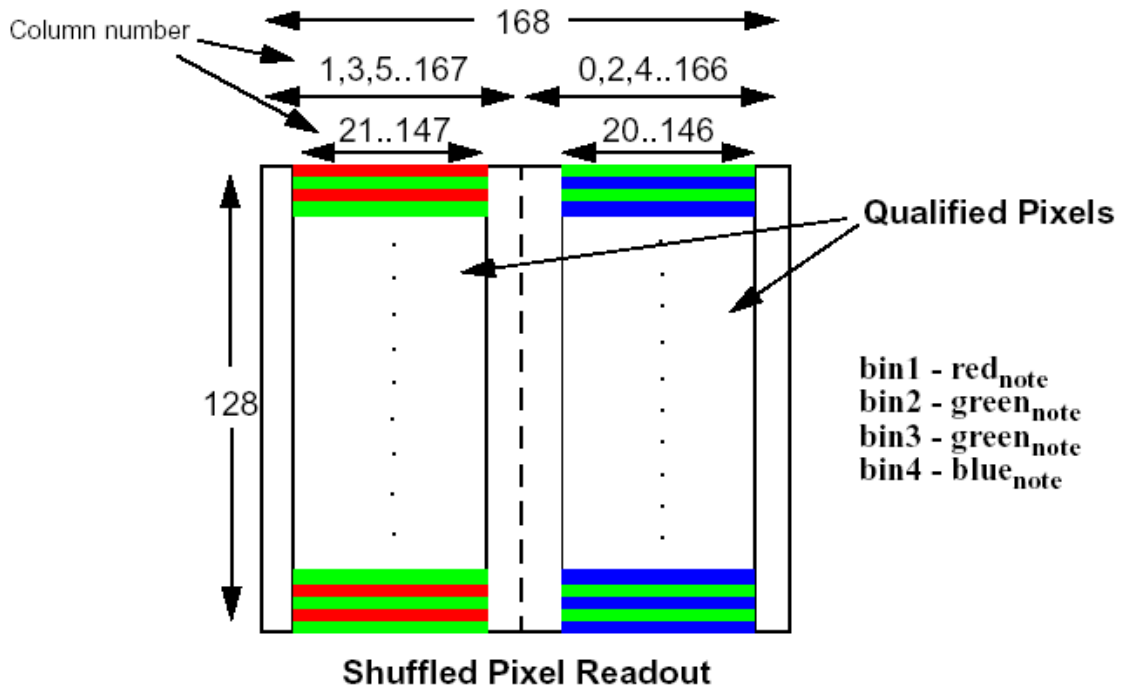
*(Typical Bayer Pattern)*

To interpolate the missing color information, nearest neighbor replication and bilinear interpolation are used. For reference, refer to [5]



(Data Readout, Source: [3])

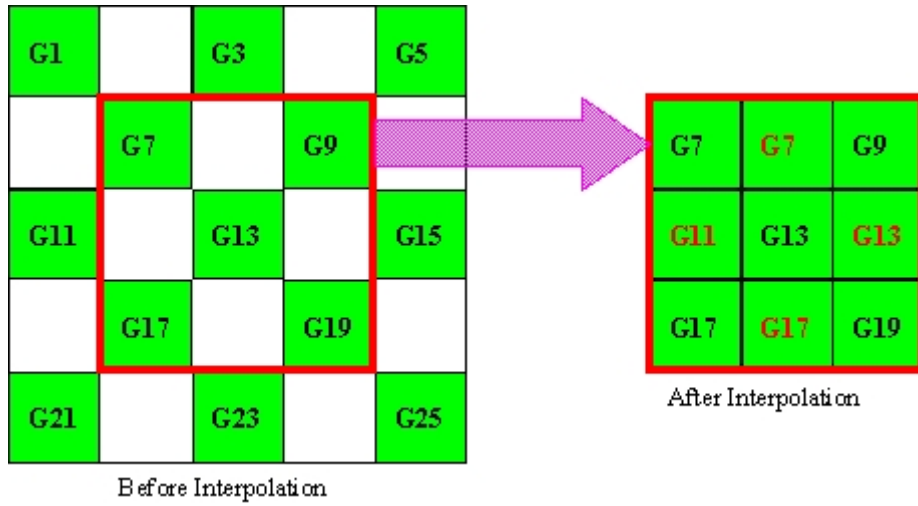
3.4.1.1. Horizontally shuffled data read-out



(Horizontally shuffled data read-out, Source:[4])

### 3.4.2. Missing Color Interpolation

The algorithm to interpolate the missing color information, used both **nearest neighbor replication**



*(Missing Color Interpolation: Nearest Neighbor Replication, Source [5])*

and **bilinear interpolation**:

G1	R2	G3	R4	G5
B6	G7	B8	G9	B10
G11	R12	G13	R14	G15
B16	G17	B18	G19	B20
G21	R22	G23	R24	G25

Bilinear Interpolation of red/blue pixels:

Interpolation of a red/blue pixel at a green position: the average of two adjacent pixel values in corresponding color is assigned to the interpolated pixel.

Example:  $B7 = (B6+B8) / 2$  and  $R7 = (R2+R12) / 2$

Interpolation of a red/blue pixel at a blue/red position: the average of four adjacent diagonal pixel values is assigned to the interpolated pixel.

Example:  $R8 = (R2+R4+R12+R14) / 4$  and  $B12 = (B6+B8+B16+B18) / 4$

### 3.4.3. Bayer Decoder, Pseudo Code:

```

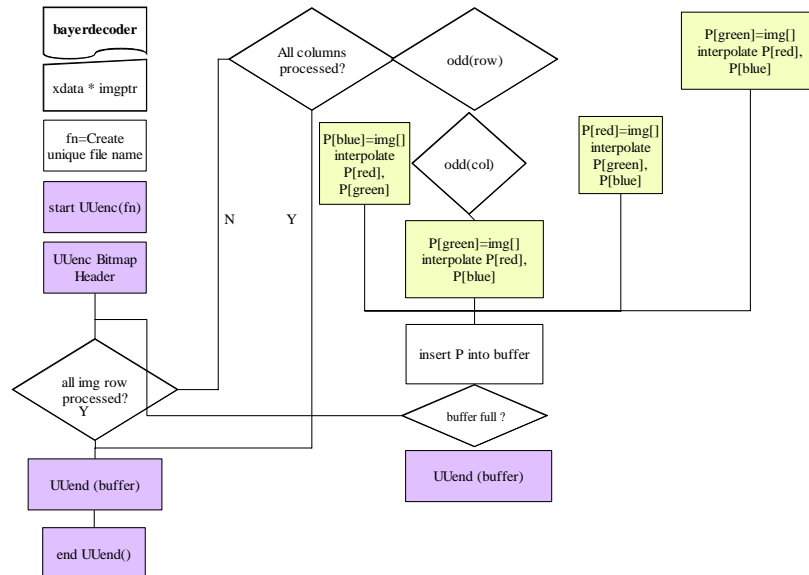
InitBuffer(10);
fn= CreateUniqueFilename();
StartUUEncoding(fn);
UUEnc( BMPHeader );

for (y=0; y<BMP_HEIGHT; y++) {

    for (x=0; x<BMP_WIDTH; x++){

        if (odd(y) {
            if (odd(x) {
                p[GREEN] = getvalue(x,y);
                p[BLUE] = interpolate(x,y);
                p[RED] = interpolate(x,y);
            } else { // even(x)
                p[z][GREEN] = interpolate(x,y);
                p[z][BLUE] = getvalue(x,y);
                p[z][RED] = interpolate(x,y);
            }
        } else {
            if (odd(x) {
                p[z][GREEN] = interpolate(x,y);
                p[z][BLUE] = interpolate(x,y);
                p[z][RED] = getvalue(x,y);
            } else {
                p[z][GREEN] = getvalue(x,y);
                p[z][BLUE] = interpolate(x,y);
                p[z][RED] = interpolate(x,y);
            }
        }
    }
}

if ((buffer full or done)
    uuencode(buffer);
}
// end_for x
} //end_for y
endEncode();
// end the encoding for this image
    
```



## 3.5. Bitmaps

### 3.5.1. Bitmap File Format

The bitmap file format supports 4-bit RLE (run length encoding), as well as 8-bit and 24-bit encoding. However, in this application, we're only dealing with the 24-bit format.

The bitmap file is divided into three sections: Bitmap file header, Bitmap information header, and Image data.

#### 3.5.1.1. Bitmap File Header

```
typedef struct tagBITMAPFILEHEADER {
    UINT bfType;
    DWORD bfSize;
    UINT bfReserved1;
    UINT bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER;
```

- **bfType:** Indicates the type of the file and is always set to BM.
- **bfSize:** Specifies the size of the whole file in bytes.
- **bfReserved1:** Reserved -- must be set to 0.
- **bfReserved2:** Reserved -- must be set to 0.
- **bfOffBits:** Specifies the byte offset from the `BitmapFileHeader` to the start of the image.

#### 3.5.1.2. Bitmap Information Header

```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
:
```

- **biSize:** Specifies the number of bytes required by the `BITMAPINFOHEADER` structure.
- **biWidth:** Specifies the width of the bitmap in pixels.
- **biHeight:** Specifies the height of the bitmap in pixels.
- **biPlanes:** Specifies the number of planes for the target device. This member must be set to 1.
- **biBitCount:** Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.
- **biCompression:** Specifies the type of compression for a compressed bitmap. In a 24-bit format, the variable is set to 0.

- **biSizeImage:** Specifies the size in bytes of the image. It is valid to set this member to 0 if the bitmap is in the `BI_RGB` format.
- **biXPelsPerMeter:** Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.
- **biYPelsPerMeter:** Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.
- **biClrUsed:** Specifies the number of color indexes in the color table actually used by the bitmap. If `biBitCount` is set to 24, `biClrUsed` specifies the size of the reference color table used to optimize performance of Windows color palettes.
- **biClrImportant:** Specifies the number of color indexes considered

**Image Data**

In the 24-bit format, each pixel in the image is represented by a series of three bytes of RGB stored as **BRG**. Each scan line is padded to an even 4-byte boundary. **The image is stored from bottom to top**, meaning that the first scan line is the last scan line in the image.

**3.6. UU-Encoding**



**Implementation in UARTex.c**

**3.6.1. Protocol**

UU-encoding is a way to code a binary, image, or any file, which may contain any characters into a standard character set that can be reliably sent over diverse networks.

**3.6.1. The Character Encoding**

The basic idea is to break groups of 3 eight-bit characters (24 bits) into 4 six-bit characters and then add 32 to each six-bit character, which maps it into the readily transmittable character.

A 6-bit value is in the range [0..63], adding 32 leads to the range: [32..95] = [0x20..0x5F], which is represented by the following ASCII characters:

! " # \$ % & ' ( ) \* + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
 @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ \_

### 3.6.2. Line Encoding

Up to 45 eight-bit characters are encoded into a single line, with a count is put at the start of the line. Therefore, in most UUEncoded files, lines start with the character "M", which is decimal 77 which, minus the 32, is 45.

### 3.6.3. File Encoding

The encoded data is directly preceded by a line containing:

*begin* <file-mode> <file-name>

The final end of encoded data is an encoded line with zero encoded characters , followed by a line containing

*end*

In both cases, the lines have no preceding or trailing blank characters. *mode* defaults to 644

### 3.6.2. Implementation

The algorithm that is used for lines in between *begin* and *end* takes three octets as input and writes four characters of output by splitting the input at six-bit intervals into four octets, containing data in the lower six bits only. These octets are converted to characters by adding a value of 0x20 to each octet, so that each octet is in the range 0x20-0x5f, and then it is assumed to represent a printable character. Where the bits of two octets are combined, the least significant bits of the first octet are shifted left and combined with the most significant bits of the second octet shifted right. Thus the three octets A, B, C are converted into the four octets:

```
D= 0x20 + (C & 0x3F);
C= 0x20 + (((B << 2) | (C >> 6)) & 0x3F);
B= 0x20 + (((A << 4) | (B >> 4)) & 0x3F);
A= 0x20 + (A >> 2);
```

## 3.7. Analysis and Conclusion

### 3.7.1. Testing and Special Results

To determine the exact layout of the downloaded raw image data, "one color" images have been taken. All red, blue, and green background have been photographed to determine the exact location of the color information in the raw image data. Moreover, pictures of thin diagonal lines have been taken, to determine the exact number of unusable columns separating the two half images.

### 3.7.2. Conclusion

Because of the limited amount of code that the evaluation version of the Keil IDE permits to link, no image enhancement algorithms could be implemented. Still, the resulting images are in a quality, allowing for simple object recognition. More cannot be expected from an inexpensive image sensor, capturing only 164 \* 126 pixels. Considering the cheap hardware and the amount of time spent, the results are surprisingly good.

Most of the development time was spent creating the fast serial driver, which enables the SDK to communicate at the required speed of 57,600 baud. Another area that took a considerable amount of time was finding the exact number of unusable data columns in the center of the image, where the two half images collide.

Finally, outputting the image uuencoded was not that much work. However, not to figure out the whole process from grabbing the image to viewing it with WinZip and MsPaint or any other bmp viewer for that matter, was definitely challenging.

## References:

- [1] Company Communication "EVK LoRes Serial Communications Protocol", Vision, 1998, RS232 command protocol specs, Jan. 1999, <http://webcam.sourceforge.net/barbie/scicomms.doc>
- [2] Jean-Pierre Dubé , "Java Tip 60: Saving bitmap files in Java", Java World Magazine. Sep 01 1998 (online version)
- [3] "Low Resolution Digital CMOS Image Sensor, VISION VV6300", VSLI Vision Limited 1998
- [4] "Low Resolution Digital CMOS Image Sensor, VV5301 & VV6301", STMicroelectronics, May 2001, <http://www.vvl.co.uk/>
- [5] "A Study of Spatial Color Interpolation Algorithms for Single-Detector Digital Cameras", Winter, 1999, Ting Chen, Information System Laboratory, Department of Electrical Engineering, Stanford University <http://ise0.stanford.edu/~tingchen/main.htm>

# Source Code

# Presentation