CARTOGRAPHER

A REPORT SUBMITTED TO DR. BELAL HOSSAIN BHUIAN OF COMPUTER SCIENCE AND ENGINEERING DEPARTMENT OF BRAC UNIVERSITY IN FULFILLMENT OF THE REQUIREMENTS FOR THESIS WORK

Istefan Islam Preetom, ID: 06210008

Tanvir Ahmed Shovon, ID: 06210019
And
Tahsin Mahmud, ID: 06210020

December 2010

Declaration

We hereby declare that this thesis is based on the results found by ourselves. Materials of work found by other researchers are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Signature of the Supervisor

8.12.10

Signature of the Author

Tahnin

Savon

ACKNOWLEDGEMENT

At first we would like to thank our supervisor Dr Mohammed Belal Hossain Bhuian for giving us the opportunity to work on this project under his supervision and also for his invaluable support and guidance throughout the period of pre-thesis and thesis semester. Through his supervision, we have learned a lot.

Lastly, we would like to thank Dr. Khalilur Rahman, Asif and Jonayet for their support and guidance in our project.

Objective

Here we described our approach on developing a mobile platform that will automatically move around and create a map of an enclosed area with possible obstacle positions. A brief explanation on why we chose this project is given afterwards. Then the discussion describes some of the similar ongoing research projects. The paper then explains about the devices we took into considerations and later on it points out the problems we faced and came up with solutions.

The purpose of this report is to show what has already been done in our project field and what we are going to achieve.

Sometimes disastrous situations occur where it becomes difficult and at times impossible for a normal human being to handle. There are also situations when human beings can't reach a place but need to gather information regarding same. To solve such difficulties computer and electrical engineers are researching on making automatic vehicles on reaching such difficult places. We can see 'mars rover' that is roaming on the planes of mars and sending us pictures from there. There are also autonomous underwater vehicles (AUVs) that can operate without human supervision. Competitions on making rescue robots are also taking place these days.

Project Overview

The vehicle consists of three motors, one of them is for the web cam movement and the other two is for - forward and backward movement and left/right movement respectively. A microcontroller is interfaced with a web cam, Darlington pairs, converter and motors. According to the given signal by the microcontroller to the motors, the motors rotate. The web cam store images as the command given to the microcontroller and the image is read. This process keeps on going unless the required work is done.

Initially the cartographer takes a picture by the web cam and if it sees a clear path, then it moves forward. If it doesn't find a suitable path then it stops and finds an alternative way. Whenever it stops reaching a desired location, the web cam rotates at an angle of 45 ° to each left and right sides by the stepper motor fixed to it, and then it again moves forward in accordance. Thus the process continues.

Thesis Progress

Two major sectors have been built:

1. Car Controller

2. Image Capture

To build these we have used microcontrollers (ATmega16 & PIC16F877A), Converters, Darlington pairs, H-Bridge, CMOS Camera and Motors as required.

Darlington Pairs

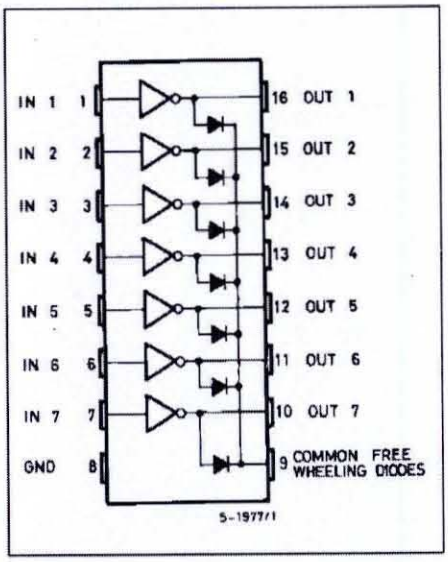
The ULN2003A is a high voltage, high current Darlington arrays each containing seven open collector Darlington pairs with common emitters. Each channel rated at 500mAand can withstand peak currents of 600mA.
Suppression diodes are included for inductive load driving and the inputs are pinned opposite the outputs to simplify board layout. The version interface to all common logic families:

ULN2001A	General Purpose, DTL, TTL, PMOS,CMOS
ULN2002A	14-25V PMOS
ULN2003A	5V TTL, CMOS
ULN2004A	6–15V CMOS, PMOS

These versatile devices are useful for driving a wide range of loads including solenoids, relays DC motors; LED displays filament lamps, thermal print heads and high power buffers.
The ULN2003A is being supplied in 16 pin plastic DIP packages with a copper Lead frame to reduce thermal resistance. They are available also in small outline package (SO-16) as ULN2001D/2002D/2003D/2004D.

Diagram Of ULN2003A:

PIN CONNECTION

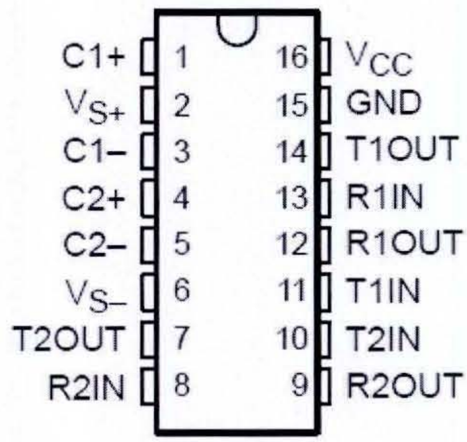


- SEVEN DARLINGTONS PER PACKAGE
- OUTPUT CURRENT 500mA PER DRIVER
- (600mA PEAK)
- OUTPUT VOLTAGE 50V INTEGRATED SUPPRESSION DIODES FOR
- INDUCTIVE LOADS OUTPUTS CAN BE PARALLELED FOR
- HIGHER CURRENT
- TTL/CMOS/PMOS/DTLCOMPATIBLE INPUTS INPUTS PINNED OPPOSITE OUTPUTS TO SIMPLIFY LAYOUT

Converter

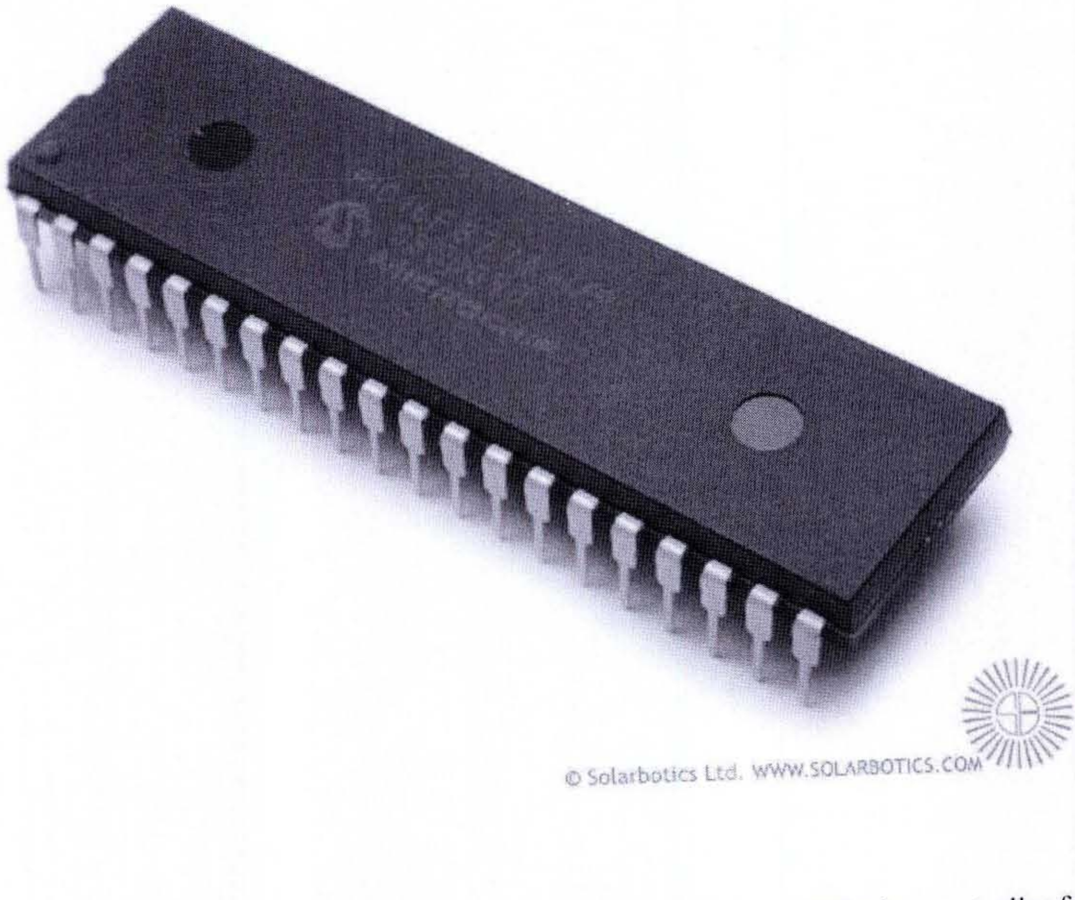
- Meet or Exceed TIA/EIA-232-F and ITU Recommendation V.28
- Operate With Single 5-V Power Supply
- Operate Up to 120 kbit/s
- Two Drivers and Two Receivers
- ±30-V Input Levels
- Low Supply Current . . . 8 mA Typical
- Designed to be Interchangeable With Maxim MAX232
- ESD Protection Exceeds JESD 22
  - 2000-V Human-Body Model (A114-A)
- Applications:
  - TIA/EIA-232-F
  - Battery-Powered Systems
  - Terminals
  - Modems
  - Computers

MAX232 . . . D, DW, N, OR NS PACKAGE
MAX232I . . . D, DW, OR N PACKAGE
(TOP VIEW)



The MAX232 is a dual driver/receiver that includes a capacitive voltage generator to supply EIA-232 voltage levels from a single 5-V supply. Each receiver converts EIA-232 inputs to 5-V TTL/CMOS levels. These receivers have a typical threshold of 1.3 V and a typical hysteresis of 0.5 V, and can accept ±30-V inputs. Each driver converts TTL/CMOS input levels into EIA-232 levels.

MICROCONTROLLER:



Here we are using a PIC16F877A microcontroller. It is the best suited microcontroller for our project. If we look at the features of it we can get a very good idea of it.

The PIC16F877A CMOS FLASH-based 8-bit microcontroller is upward compatible with the PIC16C5x, PIC12Cxxx and PIC16C7x devices. It features 200 ns instruction execution, 256 bytes of EEPROM data memory, self programming, an ICD, 2 Comparators, 8 channels of 10-bit Analog-to-Digital (A/D) converter, 2 capture/compare/PWM functions, a synchronous serial port that can be configured as either 3-wire SPI or 2-wire I2C bus, a USART, and a Parallel Slave Port.

High-Performance RISC CPU

- Lead-free; RoHS-compliant
- Operating speed: 20 MHz, 200 ns instruction cycle
- Operating voltage: 4.0-5.5V
- Industrial temperature range (-40° to +85°C)
- 15 Interrupt Sources
- 35 single-word instructions
- All single-cycle instructions except for program branches (two-cycle)

Special Microcontroller Features

- Flash Memory: 14.3 Kbytes (8192 words)
- Data SRAM: 368 bytes
- Data EEPROM: 256 bytes
- Self-reprogrammable under software control
- In-Circuit Serial Programming via two pins (5V)
- Watchdog Timer with on-chip RC oscillator
- Programmable code protection
- Power-saving Sleep mode
- Selectable oscillator options
- In-Circuit Debug via two pins

Peripheral Features

- 33 I/O pins; 5 I/O ports
- Timer0: 8-bit timer/counter with 8-bit prescaler
- Timer1: 16-bit timer/counter with prescaler
    - Can be incremented during Sleep via external crystal/clock
- Timer2: 8-bit timer/counter with 8-bit period register, prescaler and postscaler
- Two Capture, Compare, PWM modules
    - 16-bit Capture input; max resolution 12.5 ns
    - 16-bit Compare; max resolution 200 ns
    - 10-bit PWM
- Synchronous Serial Port with two modes:
    - SPI Master
    - I2C Master and Slave
- USART/SCI with 9-bit address detection
- Parallel Slave Port (PSP)
    - 8 bits wide with external RD, WR and CS controls
- Brown-out detection circuitry for Brown-Out Reset

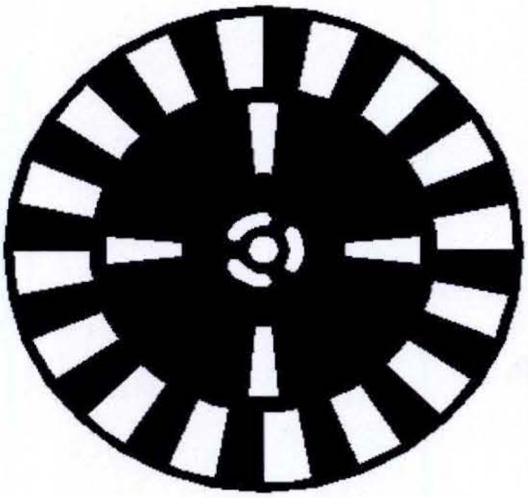
Analog Features

- 10-bit, 8-channel A/D Converter
- Brown-Out Reset
- Analog Comparator module
    - 2 analog comparators
    - Programmable on-chip voltage reference module
    - Programmable input multiplexing from device inputs and internal VREF
    - Comparator outputs are externally accessible

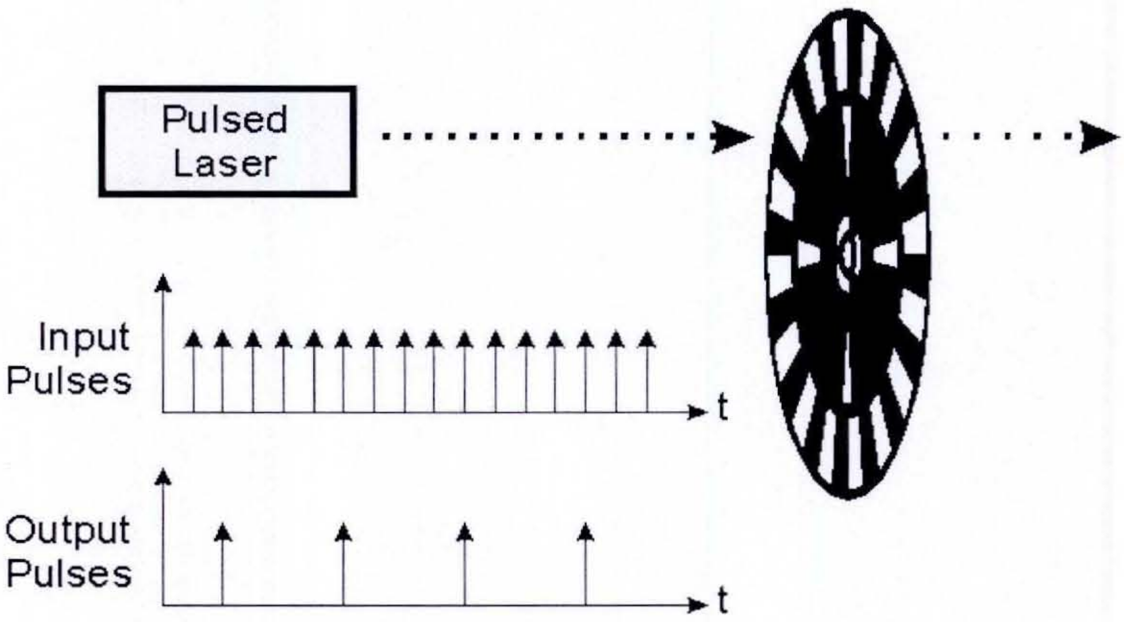
OPTICAL CHOPPER:

Pulsed laser systems working at relatively low frequencies (below say 1 kHz) often have a requirement to output every 2nd, 4th, 8th, etc pulse. There are two main considerations, the shape of the optical chopper disk required and the synchronization of the optical chopper disk with the laser pulses. Each of these problems will be considered in turn.

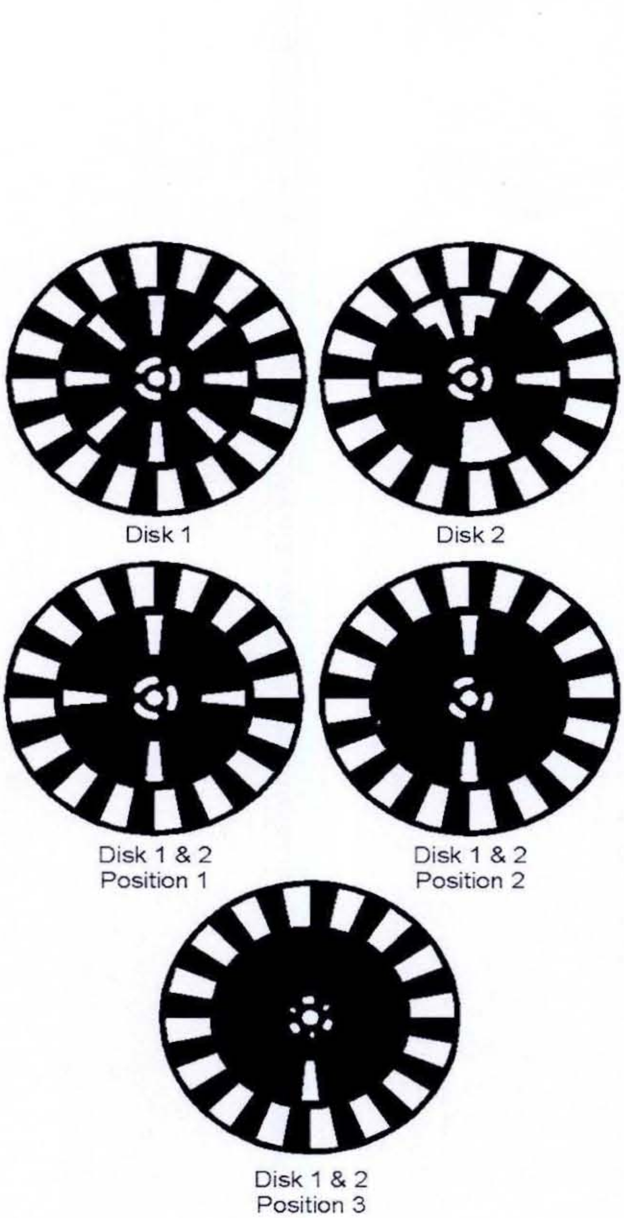
The shape required for the optical chopper disk can be very simple as shown below.



The outer set of 16 slots is used for synchronization with the laser pulses. The inner set of slots has the laser shone through it and in this case will allow through every 4th pulse as shown below.



To allow through different numbers of pulses, different disks are required. This can get expensive due to the cost of having custom disks made. Fortunately, by mounting two disks simultaneously it is possible to produce a number of different options as shown in the following picture.



Disk 1 used by itself will allow every 2nd pulse to be allowed through. Mounting disk 1 and disk 2 together on the same chopper head will allow different numbers of pulses through depending on the mutual relationship. Position 1 allows 1 in every 4 pulses through, position 2 allows through 1 in every 8 and position 3 allows through 1 in every 16.

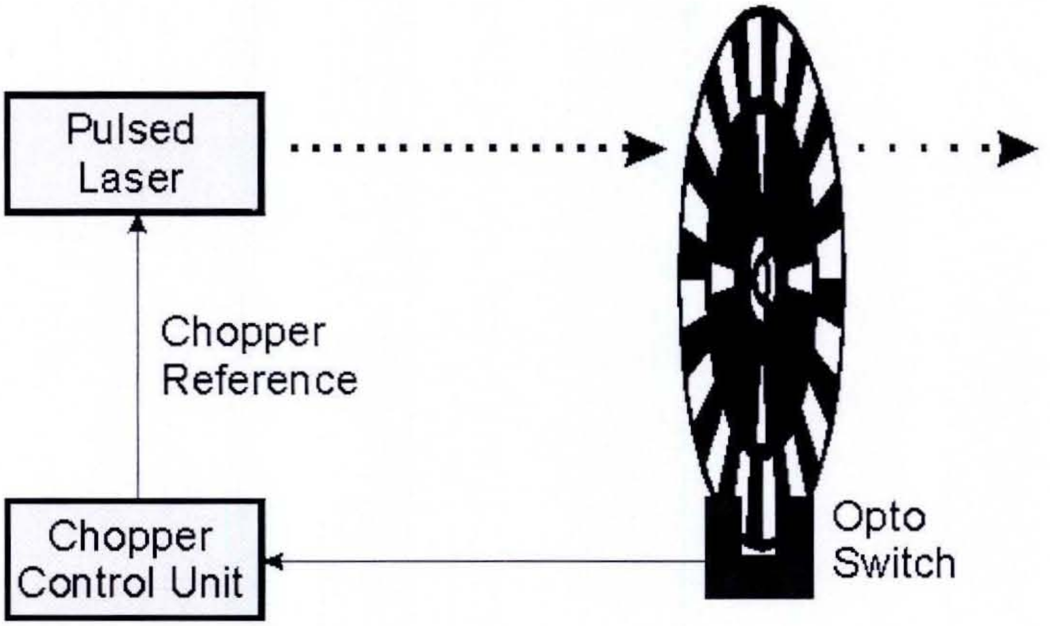
Care has been taken with design of the disks to ensure that they are balanced. Having two disks has the advantage that it is possible to have a visually unbalanced disk, such as that shown in position 3 above, that is actually balanced.

Please note that due to manufacturing tolerances, the combination of two disks in this way may not completely block light as shown. Small gaps where the disks overlap may allow a small amount of light through though this will not be a problem in a pulse picking application as these small gaps occur in between the pulses.

Synchronization of the laser pulses with the rotating chopper disk is obviously critical. There are two ways of achieving this, the laser can be synchronized to the chopper or the chopper can be synchronized to the laser.

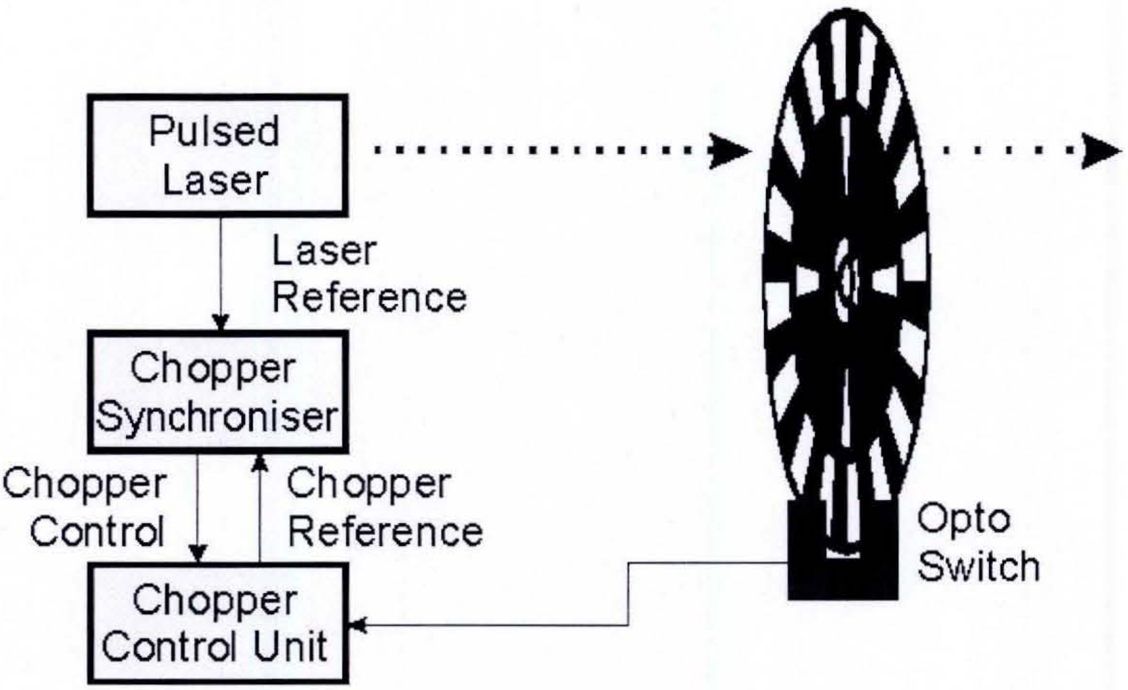
Method 1: Laser Synchronized to the Chopper

The optical chopper will generate a series of pulses from an opto-switch on the outside set of slots. This can be used to trigger the laser and is by far the simplest method of synchronization since the laser will automatically track the optical chopper as it speeds up and slows down.



Method 2: Chopper Synchronized to the Laser

Synchronizing the chopper to the laser is a lot harder than the other way around as the optical chopper is a mechanical device and can't react quickly to changes in operating frequencies. A system is required that monitors the laser pulses and the optical chopper reference and speeds up or slows down the optical chopper as required. The Scitec Instruments optical chopper synchronizer is suitable for this application.



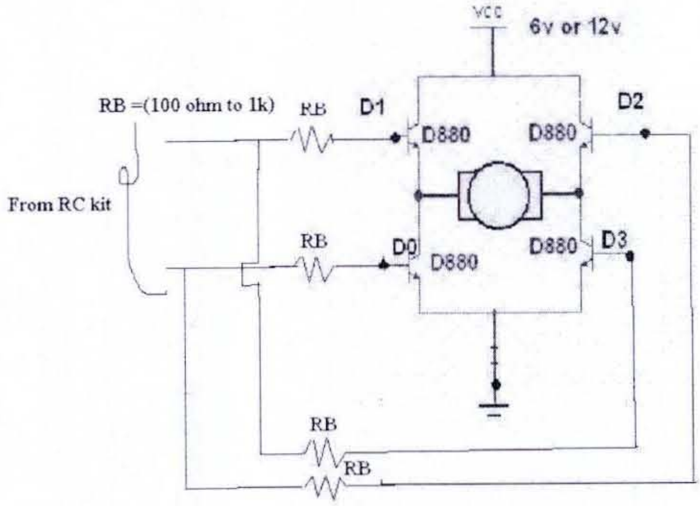
Unfortunately, this system is not perfect as it can take the system up to 10 minutes to stabilize. Jitter is also a problem being approx ±15° for the 16 slot disk shown. Method 1 is therefore recommended wherever possible.

H-BRIDGE:

This circuit drives small DC motors up to about 100 watts or 5 amps or 40 volts, whichever comes first. Using bigger parts could make it more powerful. Using a real H-bridge IC makes sense for this size of motor, but hobbyists love to do it themselves, and I thought it was about time to show a tested H-bridge motor driver that didn't use exotic parts.

Operation is simple. Motor power is required, 6 to 40 volts DC. There are two logic level compatible inputs, A and B, and two outputs, A and B. If input A is brought high, output A goes high and output B goes low. The motor goes in one direction. If input B is driven, the opposite happens and the motor runs in the opposite direction. If both inputs are low, the motor is not driven and can freely "coast", and the circuit consumes no power. If both inputs are brought high, the motor is shorted and braking occurs. This is a special feature not common to most discrete H-bridge designs, drive both inputs in most H-bridges and they self-destruct. About 0.05 amp is consumed in this state.

To do PWM(pulse width modulation) speed control, you need to provide PWM pulses. PWM is applied to one input or the other based on direction desired, and the other input is held either high("locked rotor") or low("float"). Depending on the frequency of PWM and the desired reaction of the motor, one or the other may work better for you. Holding the non-PWM'ed input low generally works best for low frequency PWM, and holding the non-PWM'ed input high generally works best at high frequencies, but is not efficient and produces a lot of heat, especially with these Darlingtons, so locked rotor is not recommended for this circuit.

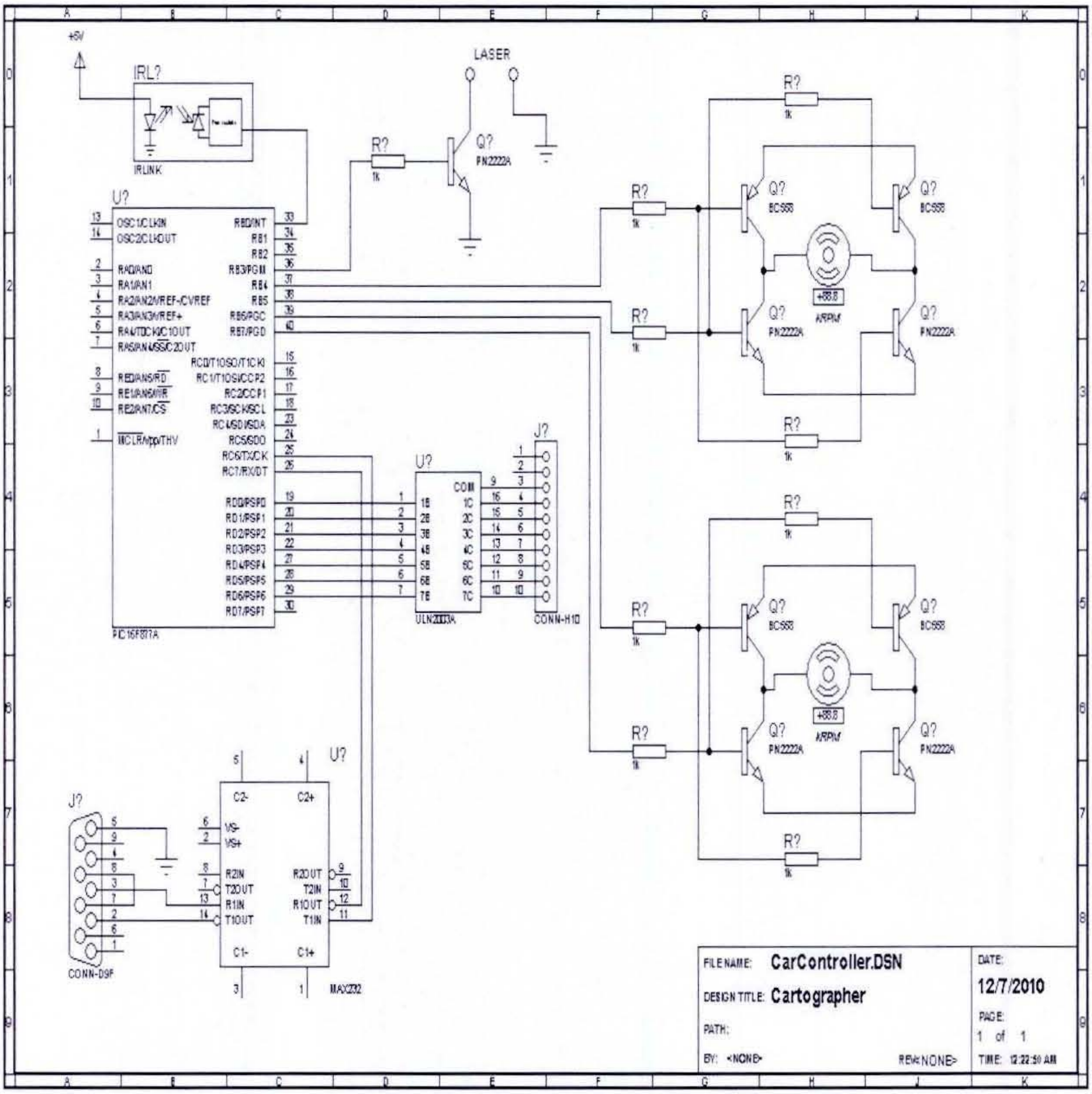


Truth table:

```
input | output
A | B | A | B
--------------
0   0 | 0   0
1   0 | 1   0
0   1 | 0   1
1   1 | 1   1
```

Here '0' mean input low or no connection '1' means input high or connected. So if A is '1' the motor will go forward and if B is '1' the motor will go backward. For breaking purpose we will switch the connection.

Overall circuit diagram for distance measurement and speed control :



General depiction

The OV6120 CMOS Image sensors are single-chip video/imaging camera devices designed to provide a high level of functionality in a single, small-footprint package. Both devices incorporate a 352 x 288 Image array capable of operating up to 60 frames per second image capture. Proprietary sensor technology utilizes advanced algorithms to cancel Fixed Pattern Noise (FPN), eliminate spreading, and drastically reduce blooming. All needed camera functions including exposure control, gamma, gain, white balance, color matrix, windowing, and more, are programmable through an SCCB (Serial Camera Control Bus) interface. Both devices can be programmed to provide image output in either 4, 8 or 16-bit digital formats. Applications include: Video Conferencing, Video Phone, Video Mail, **Still Image**, and PC Multimedia.

Features of OV6620

101,376 pixels, 1/4" lens, CIF/QCIF format

Progressive scan read out

Data format - YCrCb 4:2:2, GRB 4:2:2, RGB Raw Data

8/16 bit video data: CCIR601, CCIR656, ZV port
Wide dynamic range, anti-blooming, zero smearing

Electronic exposure / Gain / white balance control

Image enhancement - brightness, contrast, gamma, saturation, sharpness, window, etc.

Internal/external synchronization Frame exposure/line exposure option

5-Volt operation, low power dissipation
< 80 mW active power
< 10 mA in power-save mode

Gamma correction (0.45/0.55/1.00)

SCCB programmable (400 kb/s): color saturation, brightness, contrast, white balance, exposure time, gain

Getting Image from the sensor

The initial frequency of PCLK is 17.73 MHz and the ATmega16 is not fast enough to read each pixel at this frequency two solutions could be taken:

• Use additional hardware to read and store the image.

• Decrease the frequency of PCLK by writing in the register 0x11.

- Increase the frequency of at mega 16

This last solution was the one taken. The frequency taken to read the image depends on the way we read the image. If we read the image by horizontal lines we need to put the lowest frequency allowed that it is: 69,25KHz. This let us to read one line at the same time that is stored in the memory of the ATmega16. The other mode we read a vertical line of the image in each frame. In this case a higher frequency of 260KHz is used, but we need to read as many frames as vertical lines has an image to get a complete one. In the case of the horizontal lines reading the resulting image is too bright, and that is the reason why the vertical reading is used, even if we need to read as many frames as vertical lines. The horizontal reading is used to read one horizontal line and make a little image process of it. The selection of this frequencies was made experimentally trying to use the highest as possible frequency. When we want to send an image to the computer the headers and the palette are send to the computer and then we proceed to read the image from the camera. We read from the first frame the first column and send pixel by pixel with the serial port to the computer, after that the second column, and so on until we are done with the whole image.

I2C Communication with the sensor

The I2C bus is a communication protocol developed by Philips. In this protocol two pins are used, one is the clock and the other is the data. Also this protocol has a Master-Slave architecture. In our case the master is the Atmega16 and the slave the C3320 camera sensor. Registers of the sensor can be read or written by the AVR. In the writing operation the master put in the bus the writing address of device and after that put the address of the register it wants to write, and finally the byte it wants to write in the register. The reading operation is similar: first the master put in the bus the writing address of the device it wants to write, after that the register address to read from, and then the device reading address. Finally the slave puts in the bus the data requested. The I2C communication was the most difficult part of the project, because I2C protocol is not implemented hardware in the microcontroller used. Second because the C3320 camera sensor implements the SCCB protocol that it is almost the same as I2C. Three solutions were through to implement the I2C bus:

1. Use a parallel to I2C hardware converter like PCF8584. This was rejected because it will use at least 10 of the pins of the microcontroller and it will not make the software much easier.

2. Implement by software directly all the protocol.

3. Use the TWI (Two Wire Interface) present in the ATmega16, that is a synchronous bus as the I2C, and that used with some changes can implement the I2C protocol.

The last solution was the one chosen. As in the case of the **usart**, a library found in Internet was used. To test this library another I2C device was connected to the I2C bus. Once readings and writings were working in this device, the same operations were tried in the camera. The result was that the writing worked, but not the reading. After some investigations with the oscilloscope the problem was detected and solved, it was a timing problem. The read register and write register functions are implemented in the camera sensor.

Serial Communications

The communications with the computer via serial port was the first thing to be implemented because it allows to debugging by printing messages in the computer. In the computer side two text terminals were used: HyperTerminal of Windows and Real Term of the open source community. The serial port settings implemented are the next: 115200 bps, 8 bits, 1 stop bit, 0 parity bits. A velocity of 230400 bps could have been implemented in the microcontroller, but the computer cannot work with it. Because code for the serial port is widely used, the code used was based in the library Atmel AVR USART Library for GCC. The functions from the library were modified to fit the necessities of this project. These functions implement a receiving buffer: the received bytes are read by an interruption and saved in the buffer. This is the only interruption used in the system. To send bytes there is no buffer and the sending functions are blocking. We can find these functions in USART.H and USART.C.

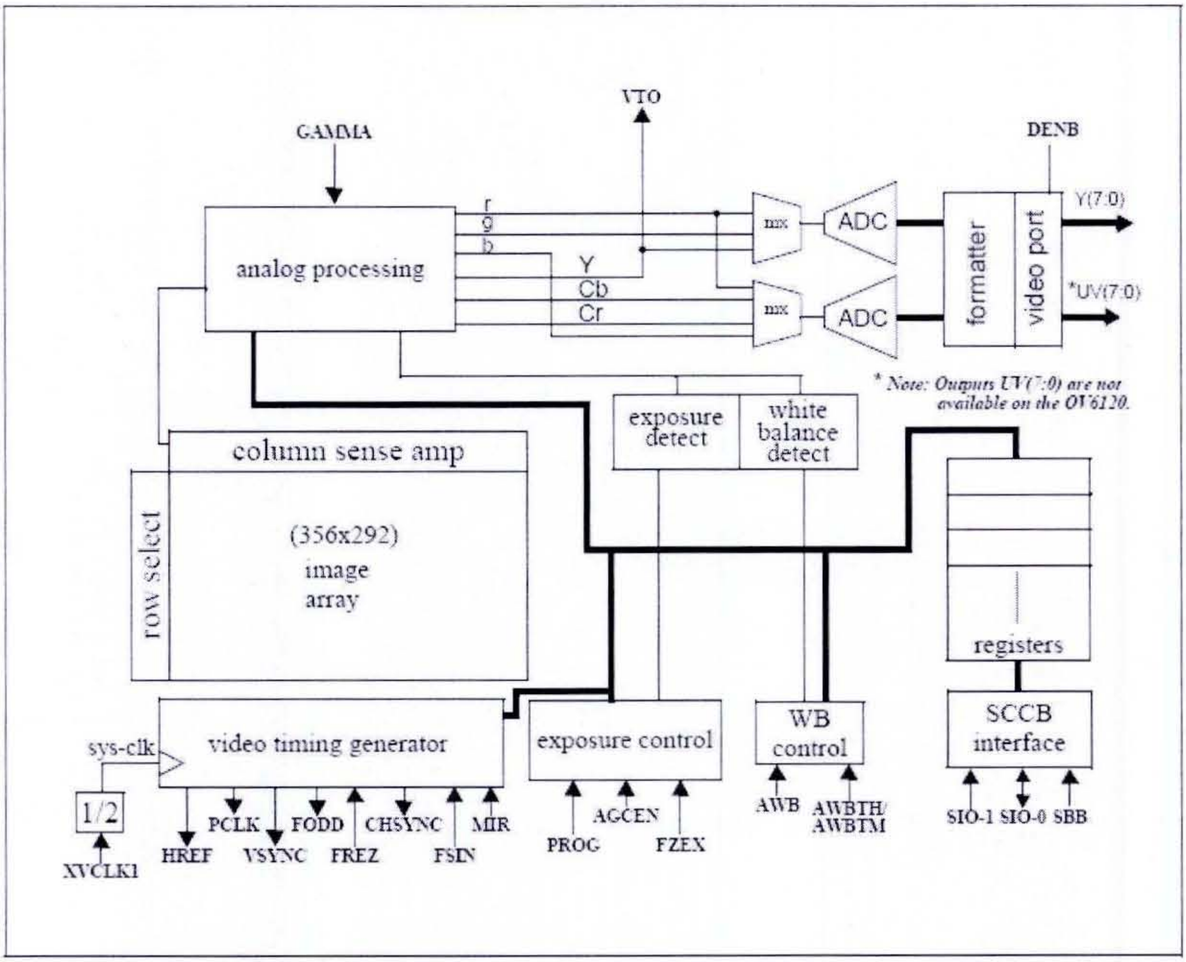


Fig-

Block Diagram of **OV6620**

OV6620 necessary pin configuration for circuit implementation:

Pin number	Function
1	SVDD Vin Array power (+5VDC)
8	AVDD Vin Analog power supply (+5VDC)
14	ADVDD Vin Analog power supply (+5VDC)
16	VSYNC/CSYS I/O Vertical sync output. At power up, read as CSYS.
18	HREF/VSFRAM I/O HREF output. At power up, read as VSFRAM
30	DGND Vin Digital ground
31	DOGND Vin Digital interface output buffer ground
32	DOVDD Vin Digital interface output buffer power supply (+5VDC)
33	PCLK/PWDB I/O PCLK output. At power up sampled as PWDB.
34	Y7/CS0 I/O Bit 7 of Y video component output. At power up, sampled as CS0.
35	Y6/CS2 I/O Bit 6 of Y video component output. At power up, sampled as CS2.
36	Y5/SHARP I/O Bit 5 of Y video component. At power up, sampled as SHARP.
37	Y4/CS1 I/O Bit 4 of Y video component. At power up, sampled as CS1
38	Y3/RGB I/O Bit 3 of Y video component output. At power up, sampled as RGB.
40	Y1 I/O Bit 1 of Y video component output.
41	Y0/CBAR I/O Bit 0 of Y video component output. At power up, sampled as CBAR
45	SIO-1 I SCCB serial interface clock input
46	SIO-0 I/O SCCB serial interface data input and output.
48	SGND Vin Array ground
39	Y2/G2X I/O Bit 2 of Y video component output. At power up, sampled as G2X.

Configuring the OV6620 Image Sensors

Two methods are provided for configuring the OV6620 ICs for specific application requirements. At power up, the OV6620 sensors read the status of certain pins to determine what, if any, power up default settings are requested. Once the reading of the external pins is completed, the device configures its internal registers according to the specified pins. Not all device functions are available for configuration through external pin.

Depiction of the work

The overall communication is done by I2C protocol. We have used SIO-0 and SIO-1 and Y1,Y2,Y3,Y4,Y5,Y6,Y7 for capturing image.Y1-Y7 pins are used for capturing gray scale image we can also use VRCAP-1 and VRCAP-3 for capturing color photo. There is another pin which can be used for capturing image at dark and that is VSYCC pin. We have used CMYK color format and BMP as picture format. Other PINS are used for RGB formatting of captured image. SCL of camera Sensor and Pc's SCL as clock matching. Than Microcontroller's SDA and sensor's SDA will be shorted .We used Max232 for serial communication with pc.It also convert pc's 12 V to 5v to fed lens and microcontroller as Atmega16 and sensor only can operate at 5V. The on-chip 8-bit A-to-D converters operate at up to 9 MHz, fully synchronous to the pixel rate. Actual conversion rate is set as a function of the frame rate. A-to-D black-level calibration circuitry ensures the following:

– the black level of Y/CMYK is normalized to a value of 16
– the peak white level is limited to 240
– CrCb black level is 128
– Peak/Bottom is 240/16
– RGB raw data output range is 16/240

But The ATmega16 operates up to 8Mhz and the OV6620 Operates up to 17.17 MHZ. So we had to add a Crystal Oscillator to speed up the operating frequency of Atmega16 up to 17.73Mhz. We ware able to generate 16Mhz by using crystal oscillator which was fine to get data. After building up the circuit we connect it with Computers Serial communication Port. Than we Connect to HyperTerminal function from of the PC. Than We select

Bits per sec to 19200
Data Bit-8
Parity None
Stop Bits-1
Flow control -None

Thus the circuit got connected to the serial Bus of PC.
Now we Used Terminal V19B to check whether the OV6220 Sensor is sensing image or not.
We were able to see some bits streaming throw the lens.
The YCrCb/RGB Raw Data signal from the analog processing section is fed to two on-chip 8-bit Analog-to-Digital (A-to-D) converters: one for the Y channel and one shared by the CrCb/ channels. The A-to-D converted data stream is further conditioned in the digital formatter. The processed signal is delivered to the digital video port through the video multiplexer which routes the user-selected 16-, 8-, or 4-bit video data the correct output pins.

Microcontroller

We used Atmega16 microcontroller to operate the system. This microcontroller has built in 16Kb programmable flash memory moreover it has the ability to read raw data and conversion ability which can convert the sensor image to bmp format and USART port as our whole communication process is based on serial communication this is the basic region to choose Atmega16.Basic features and used pin are described here
SCL and SDA Pins These pins interface the AVR TWI with the rest of the MCU system. The output drivers contain a slew-rate limiter in order to conform to the TWI specification. The input stages contain a spike suppression unit removing spikes shorter than 50 ns. Note that the internal pull-ups in the AVR Pads can be enabled by setting the PORT bits corresponding to the SCL and SDA pins, as explained in the I/O Port section of Atmega16 data sheet. The internal pull-ups can in some systems eliminate the need for external ones.

Bit Rate Generator
Unit
This unit controls the period of SCL when operating in a Master mode. The SCL period is controlled by settings in the TWI Bit Rate Register (TWBR) and the Prescaler bits in the TWI Status Register (TWSR). Slave operation does not depend on Bit Rate or Prescaler settings, but the CPU clock frequency in the slave must be at least 16 times higher than the SCL frequency. Note that slaves may prolong the SCL low period, thereby reducing the average TWI bus clock period. The SCL frequency is generated according to the following equation:

SCL frequency= CPU Clock frequency/16 + 2(TWBR)*4*TWPS*

- TWBR = Value of the TWI Bit Rate Register
- TWPS = Value of the prescaler bits in the TWI Status Register

Note: Pull-up resistor values should be selected according to the SCL frequency and the capacitive bus line load.

TWI Bit Rate Register
– TWBR
• Bits 7..0 – TWI Bit Rate Register
TWBR selects the division factor for the bit rate generator. The bit rate generator is a frequency divider which generates the SCL clock frequency in the Master modes.

TWI Control Register –
TWCR

The TWCR is used to control the operation of the TWI. It is used to enable the TWI, to initiate a master access by applying a START condition to the bus, to generate a receiver acknowledge, to generate a stop condition, and to control halting of the bus while the data to be written to the bus are written to the TWDR. It also indicates a write collision if data is attempted written to TWDR while the register is inaccessible.

• Bit 7 – TWINT: TWI Interrupt Flag

This bit is set by hardware when the TWI has finished its current job and expects application software response. If the I-bit in SREG and TWIE in TWCR are set, the MCU will jump to the TWI interrupt Vector. While the TWINT Flag is set, the SCL low period is stretched. The TWINT Flag must be cleared by software by writing a logic one to it. Note that this flag is not automatically cleared by hardware when executing the interrupt routine. Also note that clearing this flag starts the operation of the TWI, so all accesses to the TWI Address Register (TWAR), TWI Status Register (TWSR), and TWI Data Register (TWDR) must be complete before clearing this flag.

• Bit 6 – TWEA: TWI Enable Acknowledge Bit

The TWEA bit controls the generation of the acknowledge pulse. If the TWEA bit is written to one, the ACK pulse is generated on the TWI bus if the following conditions are met:

1. The device's own slave address has been received.
2. A general call has been received, while the TWGCE bit in the TWAR is set.
3. A data byte has been received in Master Receiver or Slave Receiver mode. By writing the TWEA bit to zero, the device can be virtually disconnected from the Two-wire Serial Bus temporarily. Address recognition can then be resumed by writing the TWEA bit to one again.

• **Bit 5 – TWSTA: TWI START Condition Bit**

The application writes the TWSTA bit to one when it desires to become a master on the Two wire Serial Bus. The TWI hardware checks if the bus is available, and generates a START condition on the bus if it is free. However, if the bus is not free, the TWI waits until a STOP condition is detected, and then generates a new START condition to claim the bus Master status. TWSTA must be cleared by software when the START condition has been transmitted.

• **Bit 4 – TWSTO: TWI STOP Condition Bit**

Writing the TWSTO bit to one in Master mode will generate a STOP condition on the Two-wire Serial Bus. When the STOP condition is executed on the bus, the TWSTO bit is cleared automatically. In slave mode, setting the TWSTO bit can be used to recover from an error condition. This will not generate a STOP condition, but the TWI returns to a well-defined unaddressed
slave mode and releases the SCL and SDA lines to a high impedance state.

• **Bit 3 – TWWC: TWI Write Collision Flag**

The TWWC bit is set when attempting to write to the TWI Data Register – TWDR when TWINT is low. This flag is cleared by writing the TWDR Register when TWINT is high.

• **Bit 2 – TWEN: TWI Enable Bit**

The TWEN bit enables TWI operation and activates the TWI interface. When TWEN is written to one, the TWI takes control over the I/O pins connected to the SCL and SDA pins, enabling the
slew-rate limiters and spike filters. If this bit is written to zero, the TWI is switched off and all TWI transmissions are terminated, regardless of any ongoing operation.

• **Bit 1 – Res: Reserved Bit**

This bit is a reserved bit and will always read as zero.

• **Bit 0 – TWIE: TWI Interrupt Enable**

When this bit is written to one, and the I-bit in

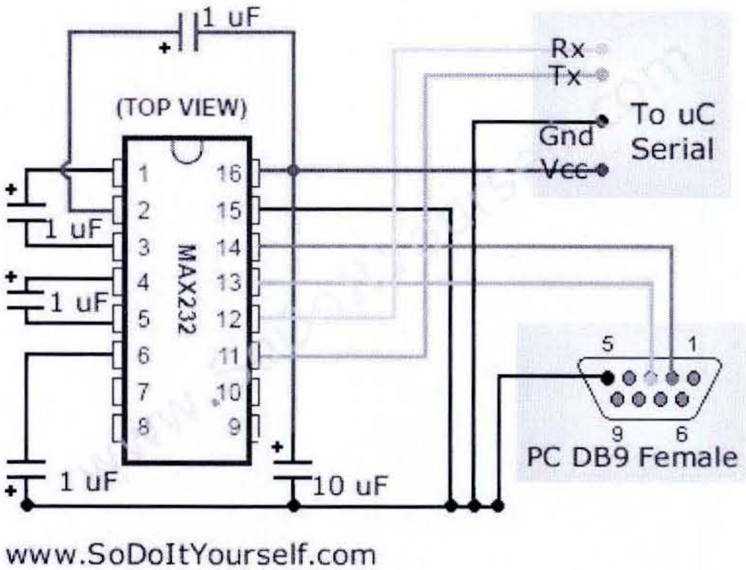
BMP formation

BMP file contain 4 group of data structure

1. BMP File Header Stores general information about the BMP file.
2. Bitmap Information Stores detailed information about the bitmap image.
3. Color Palette Stores the colors use for indexed color bitmaps. This is for 1,4,8 bits per pixel.
4. Bitmap Data Stores the actual image, pixel by pixel.

Max232 IC

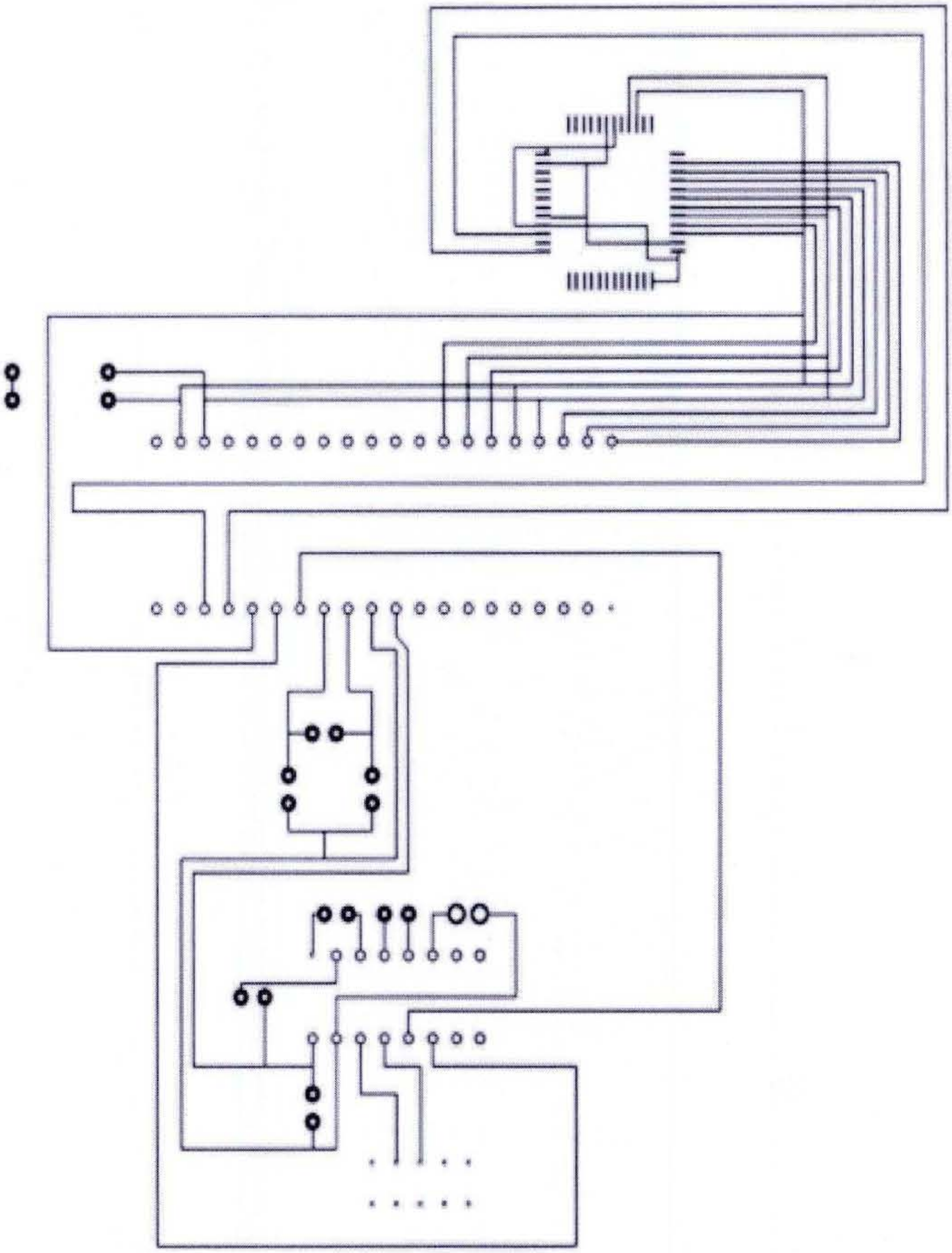
This IC is used for serial communication between microcontroller and pc or wireless device Rf module RST-tx. It contains two transmitters and receivers port to communicate with pc and microcontroller

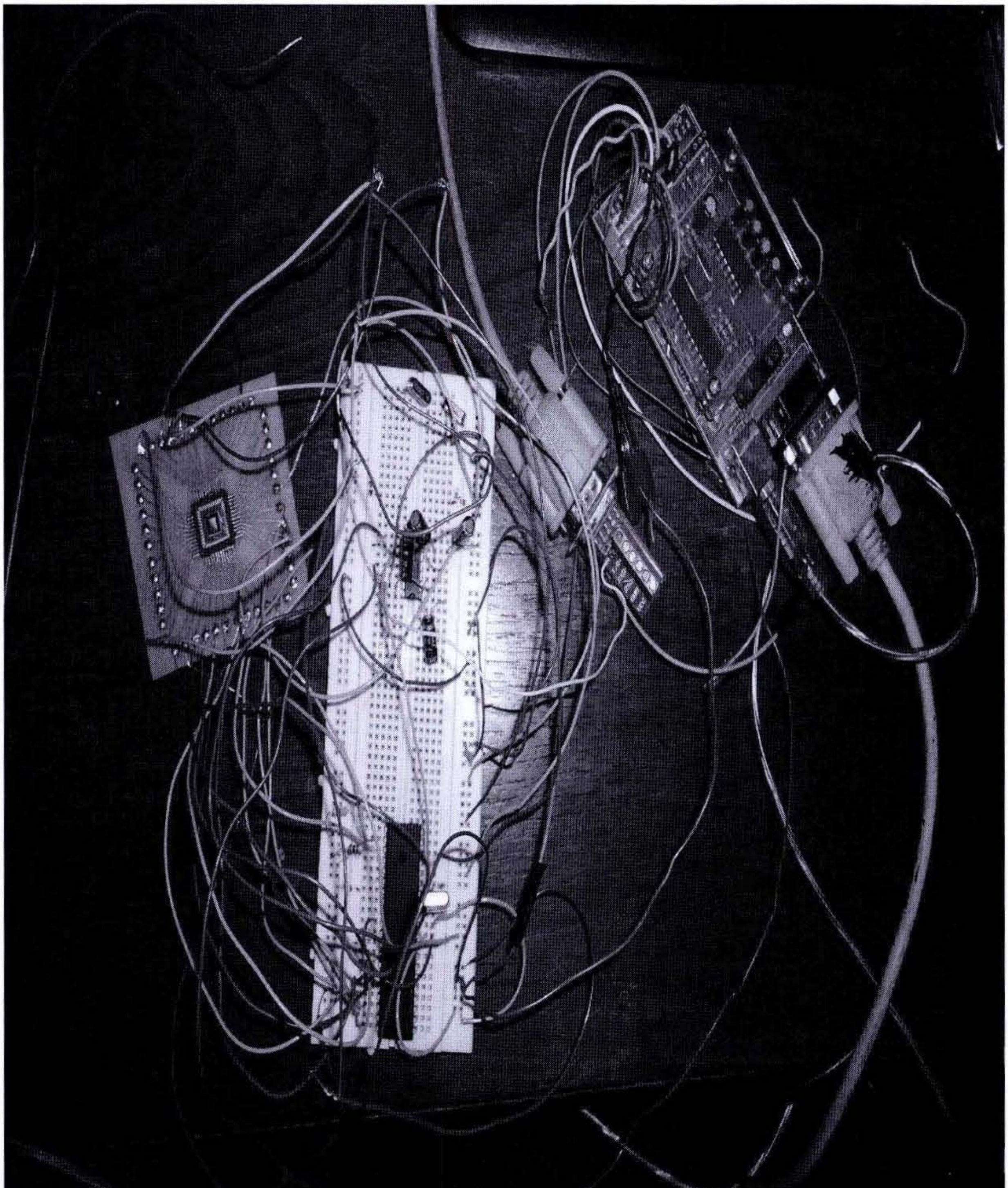


Other Apparatus Used in Circuit

Crystal oscillator: It is used for pull up the microcontroller frequency to mach with camera sensor's frequency
Resistor: We used four resistors of 10k ohm
Capacitor: Connected with max232, here five capacitors of 1micro fared used

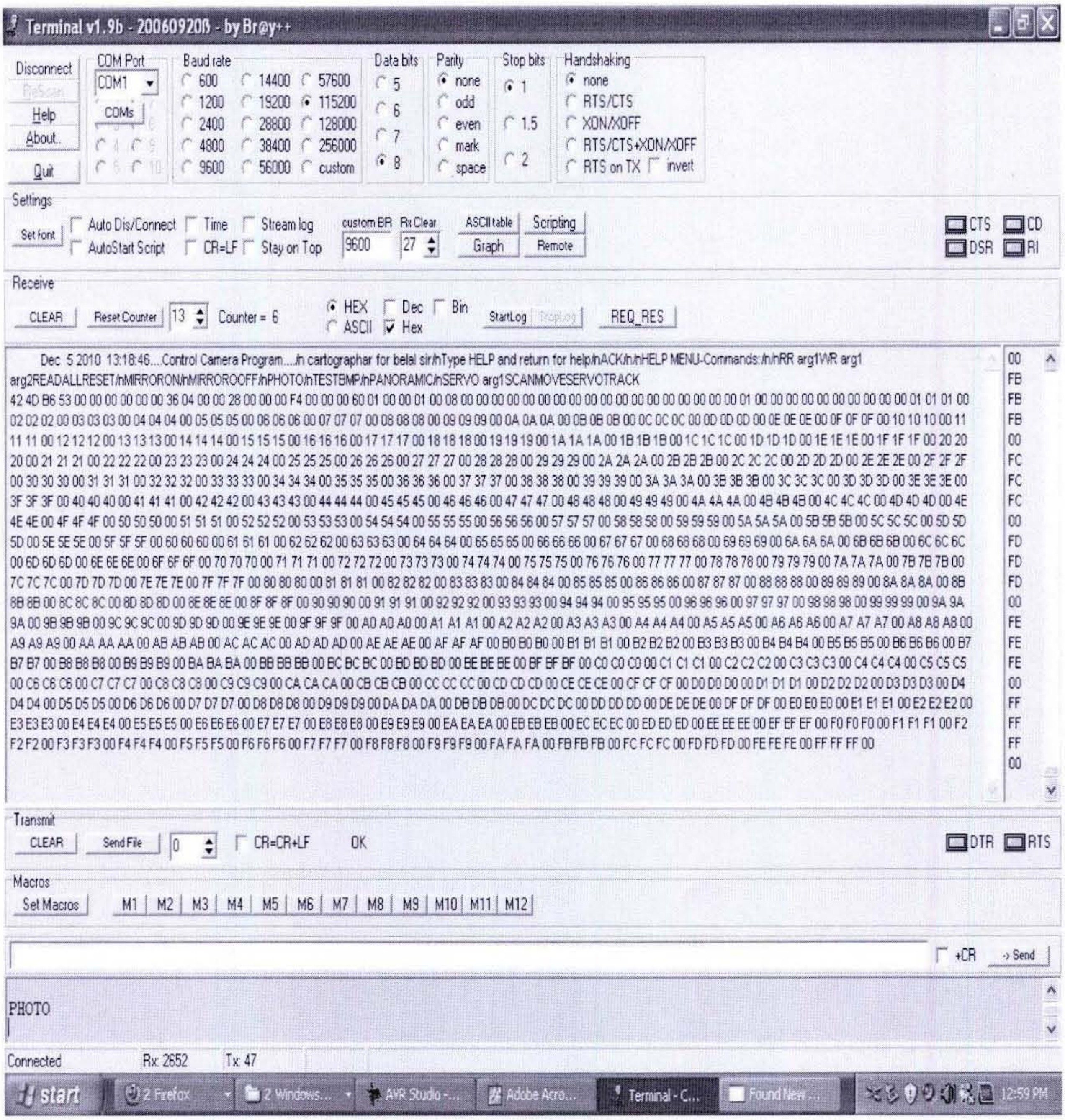
Overall circuit design in pcb layout





This the real image of our work.

Output taken from cmos camera sensor :



Here the out put shows only Hexadecimal value because the software we used for serial communication is unable to show picture. To get a complete picture we need to use this value for image processing which can be done by MATLAB simulation or by C++ programming. For time scarcity we cant complete this task. As

Conclusion

The purpose of this report is to show what has already been done in our project field and what we are going to achieve. In this context we thought of our current perspectives and motivated in implementing similar featured platform with its own area mapping and obstacle detection capability to handle dangerous and difficult situations. In this paper we presented the devices we need such as microcontrollers and others to work out.
Thus we will be able to implement it in the future. One of the major parts that is wirelessly data transfer and image capturing wirelessly can be implemented in the future work.

It is concluded that further research and experiments with new devices that are being developed every day, will help us improve and optimize our current project to be more accurate and precise. It is also suggested that a successful implementation of our project will help in handling situations that are difficult in normal context.

Future Work:

- ☐ Implement heat sensor
- ☐ Integrate with robotic arm
- ☐ Integrate Artificial Intelligence.
- ☐ Implementation of Wireless Communication

References

[1] Cosmanescu A, Miller B, Magno T, Ahmed A, Kremenic I. *Design and implementation of a wireless (Bluetooth [registered trademark]) four channel bio-instrumentation amplifier and digital data acquisition device with userselectable gain, frequency, and driven reference*, EMBS Annual International Conference, IEEE Sep. 3, 2006.

[2] McDermott-Wells P. *What is Bluetooth?*, Potentials, IEEE 2005; 23(5): 33-35.

[3] Advantages of stepper motor. *http://www.sapiensman.com/ESDictionary/docs/d6.htm*

[4] Microcontroller specification.
http://www.atmel.com/dyn/resources/prod_documents/doc2503.pdf

[5] USART. *http://www.ip-extreme.com/downloads/usart_brochure_080121.pdf*

[6] http://www.radio.gov.uk/topics/conformity/conform-index.htm

[7] http://www.ai.sri.com/people/flakey/control.html

[8] Hee Chang Moon; Kyoung Moo Min; Jung Ha Kim; *Vision system of Unmanned Ground Vehicle.*

[9] Madhavan, R.; Schlenoff, C. *The effect of process models on short-term prediction of moving objects for unmanned ground vehicles.*

[10] Jong Hoon Ahnn, *Project Title*: *Robot control using the wireless communication and the serial communication*

INDEX

```
#include "bmp.h"

void createheader(char *header, int heigh, int width){
  char *p;
  char bytelow;
  char bytehigh;
  int sizefile = 1078 +(heigh*width);
  bytelow = sizefile & 0x00FF;
  bytehigh = (sizefile & 0xFF00)>>8;
  p = header;

  //2 Bytes --BM Starting
  *p = 'B';
  p++;
  *p = 'M';
  p++;
  //4 Bytes -- Size of file in bytes = 14 + 40 +1024 + HEIGH * WIDTH
(100*100) = 2078 = 0x049A
  *p = bytelow;
  p++;
  *p = bytehigh;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  //4 Bytes of reserved (= 0)
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  //4 Bytes of offset to the init of the data
  *p = 0x36;
  p++;
  *p = 0x04;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;

}

void createinfoheader(char *infoheader, int heigh, int width){
  char *p;
```

```
char heighlow;
char heighhigh;
char widthlow;
char widthhigh;

heighlow = heigh & 0x00FF;
heighhigh = (heigh & 0xFF00)>>8;
widthlow = width & 0x00FF;
widthhigh = (width & 0xFF00)>>8;

p = infoheader;

//4 Bytes -- Size of InfoHeader =40
*p = 40;
p++;
*p = 0;
p++;
*p = 0;
p++;
*p = 0;
p++;
//4 Bytes -- specifies the width of the image, in pixels.
*p = widthlow;
p++;
*p = widthhigh;
p++;
*p = 0;
p++;
*p = 0;
p++;
//4 Bytes -- specifies the heigth of the image, in pixels.
*p = heighlow;
p++;
*p = heighhigh;
p++;
*p = 0;
p++;
*p = 0;
p++;
//2 Bytes -- Number of planes of the image
*p = 1;
p++;
*p = 0;
p++;
//2 Bytes Bits per Pixel  -- In our case 8.
*p = 8;
p++;
*p = 0;
p++;
//4 bytes -- Type of Compression 0 = BI_RGB   no compression
*p = 0;
p++;
*p = 0;
p++;
*p = 0;
p++;
*p = 0;
```

```
  p++;
  //4 bytes -- ImageSize  (compressed) It is valid to set this =0
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  //XpixelsPerM 4 bytes horizontal resolution: Pixels/meter
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  //YpixelsPerM 4 bytes vertical resolution: Pixels/meter
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  //ColorsUsed 4 bytes Number of actually used colors =256
  *p = 0;
  p++;
  *p = 1;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  //ColorsImportant 4 bytes Number of important colors  0 = all
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
  *p = 0;
  p++;
}

void usart_putnumchars(char *header, int num){
  char *p;
  p = header;
  for(int i=0; i<num;i++){
    usart_putc(*p);
    p++;
  }

}
```

```
void sendtable(void){
  for(int i=0; i<256;i++){
    usart_putc(i);
      usart_putc(i);
      usart_putc(i);
      usart_putc(0);
  }

}
void senddata(void){
  for(int i=0; i<244;i++){
    for(int j=0; j<44;j++){
    usart_putc(0+5*j);
      usart_putc(10+5*j);
      usart_putc(20+5*j);
      usart_putc(0+2*i);
      usart_putc(50+2*i);
      usart_putc(0+9*j);
      usart_putc(10+9*j);
      usart_putc(20+9*j);
      }
  }

}
```

```
#include "cam.h"
#include "delay.h"
#include "usart.h"
#include "servo.h"

void camports_init(void){
  DDRY = 0x00;
  DDRD = (DDRD & 0xE3);
}

void photo(void){
      for(int y = 0; y<352; y++){
        while(isVSYNup);
        while(isVSYNdown);
        for(int r = 0;r<244;r++){
          while(isHREFdown);
            for(int h = 0;h<y;h++){
                    while(isPCLKup);
              while(isPCLKdown);
          }
            usart_putc(PINY);
            while(isHREFup);
        }
      }
```

```
}

void panoramic(void){
    for(int x = -950; x<=950; x++){
        set_servo_pos(-x);
        Delay_1ms(30);
        while(isVSYNup);
        while(isVSYNdown);
        for(int r = 0;r<244;r++){
          while(isHREFdown);
            for(int h = 0;h<176;h++){  // I take the center column
                    while(isPCLKup);
              while(isPCLKdown);
          }
            usart_putc(PINY);
            while(isHREFup);
        }
      }
}

int getcenter(char *row){
    int maxvalue = 0;
      int maxpos = 0;
      int previous = 0;
      int center = -1;

      char *p;
    p = row;

      while(isVSYNup);
      while(isVSYNdown);
      for(int r = 0;r<20;r++){    // I wait for row 20
      while(isHREFup);
      while(isHREFdown);
      }
      while(isHREFup){
        while(isPCLKdown);
        *p = PINY;
        p++;
        while(isPCLKup);
        while(isPCLKdown);
        while(isPCLKup);
        while(isPCLKdown);
        while(isPCLKup);
        while(isPCLKdown);
        while(isPCLKup);
      }

      p = row;
      for(int i = 1;i<=88;i++){
        if(*p >= 225){
          previous++;
            if (previous>=maxvalue){
              maxvalue = previous;
              maxpos=i;
            }
        } else previous = 0;
```

```
        p++;
      }
      if(maxvalue>1) center = (int) maxpos - (maxvalue/2);
      return center;
}
```

```
#include "delay.h"

void Delay_100us(unsigned char t) {
  unsigned int i;
  if (t==0) return;
  while (t--) for(i=0;i<K_DELAY_100us; i++);
}

void Delay_1ms(unsigned char t) {
  unsigned int i;
  if (t==0) return;
  while (t--) for(i=0;i<K_DELAY_1ms; i++);
}

void Delay_10ms(unsigned char t) {
  unsigned int i;
  if (t==0) return;
  while (t--) for(i=0;i<K_DELAY_10ms; i++);
```

```
#include <avr/io.h>
#define F_CPU 16000000

#define K_DELAY_100us   F_CPU/61349
#define K_DELAY_1ms           F_CPU/6013
#define K_DELAY_10ms    F_CPU/600

void Delay_100us(unsigned char t);
void Delay_1ms(unsigned char t);
void Delay_10ms(unsigned char t);
```

```
include "I2C_CAM.h"


void
i2c_init(int clk)
{

  /* initialize TWI clock: 100 kHz clock, TWPS = 0 => prescaler = 1 */
#if defined(TWPS0)
```

```
  /* has prescaler (mega128 & newer) */
  TWSR = 0;
#endif
  TWBR = (clk*1000000 / 100000UL - 16) / 2;

}

int
i2c_read_bytes(uint16_t eeaddr, int len, uint8_t *buf)
{

  unsigned long int counter=0;
  uint8_t sla, twcr, n = 0;
  int rv = 0;


  /* patch high bits of EEPROM address into SLA */
  sla = TWI_SLA_CAM |(((eeaddr >> 8) & 0x07) << 1);

  /*
   * Note [6]
   * First cycle: master transmitter mode
   */
  restart:
  if (n++ >= MAX_ITER)
    return -1;
  begin:

  TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); /* send start condition
*/
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_REP_START:          /* OK, but should not happen */
    case TW_START:
      break;

    case TW_MT_ARB_LOST:        /* Note [7] */
      goto begin;

    default:
      return -1;                /* error: not in start condition */
                                /* NB: do /not/ send stop condition */
    }


   /* Note [8] */
  /* send SLA+W */
  TWDR = sla | TW_WRITE;
  TWCR = _BV(TWINT) | _BV(TWEN); /* clear interrupt to start
transmission */
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_MT_SLA_ACK:
      break;
```

```
    case TW_MT_SLA_NACK:        /* nack during select: device busy
writing */
                                /* Note [9] */
      goto restart;

    case TW_MT_ARB_LOST:        /* re-arbitrate */
      goto begin;

    default:
      goto error;               /* must send stop condition */
    }

  TWDR = eeaddr;                /* low 8 bits of addr */
  TWCR = _BV(TWINT) | _BV(TWEN); /* clear interrupt to start
transmission */
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_MT_DATA_ACK:
      break;

    case TW_MT_DATA_NACK:
      goto quit;

    case TW_MT_ARB_LOST:
      goto begin;

    default:
      goto error;               /* must send stop condition */
    }


  TWCR = 0; // Stop the twi interface to make the camera able to
rescognise the new start
  while (counter != 0x0020)
          {
               counter++;
          }


  /*
   * Note [10]
   * Next cycle(s): master receiver mode
   */
  TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); /* send (rep.) start
condition */
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_START:              /* OK, but should not happen */
    case TW_REP_START:
      break;

    case TW_MT_ARB_LOST:
      goto begin;
```

```
    default:
      goto error;
    }

  /* send SLA+R */
  TWDR = (sla | TW_READ);
  TWCR = _BV(TWINT) | _BV(TWEN); /* clear interrupt to start
transmission */
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_MR_SLA_ACK:
      break;

    case TW_MR_SLA_NACK:
      goto quit;

    case TW_MR_ARB_LOST:
      goto begin;

    default:
      goto error;
    }

  for (twcr = _BV(TWINT) | _BV(TWEN) | _BV(TWEA) /* Note [11] */;
       len > 0;
       len--)
    {
      if (len == 1)
        twcr = _BV(TWINT) | _BV(TWEN); /* send NAK this time */
      TWCR = twcr;              /* clear int to start transmission */
      while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
      switch ((TW_STATUS))
        {
        case TW_MR_DATA_NACK:
          len = 0;              /* force end of loop */
          /* FALLTHROUGH */
        case TW_MR_DATA_ACK:
          *buf++ = TWDR;
          rv++;
          break;

        default:
          goto error;
        }
    }
  quit:
  /* Note [12] */
  TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN); /* send stop condition */

  return rv;

  error:
  rv = -1;
  goto quit;
}
```

```
int
i2c_write_page(uint16_t eeaddr, int len, uint8_t *buf)
{
  uint8_t sla, n = 0;
  int rv = 0;
  uint16_t endaddr;

  if (eeaddr + len < (eeaddr | (PAGE_SIZE - 1)))
    endaddr = eeaddr + len;
  else
    endaddr = (eeaddr | (PAGE_SIZE - 1)) + 1;
  len = endaddr - eeaddr;

  /* patch high bits of EEPROM address into SLA */
  sla = TWI_SLA_CAM | (((eeaddr >> 8) & 0x07) << 1);

  restart:
  if (n++ >= MAX_ITER)
    return -1;
  begin:

  /* Note 13 */
  TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); /* send start condition
*/
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_REP_START:           /* OK, but should not happen */
    case TW_START:
      break;

    case TW_MT_ARB_LOST:
      goto begin;

    default:
      return -1;                 /* error: not in start condition */
                                 /* NB: do /not/ send stop condition */
    }

  /* send SLA+W */
  TWDR = sla | TW_WRITE;
  TWCR = _BV(TWINT) | _BV(TWEN); /* clear interrupt to start
transmission */
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_MT_SLA_ACK:
      break;

    case TW_MT_SLA_NACK:         /* nack during select: device busy
writing */
      goto restart;

    case TW_MT_ARB_LOST:         /* re-arbitrate */
      goto begin;

    default:
```

```
      goto error;                 /* must send stop condition */
    }

  TWDR = eeaddr;                  /* low 8 bits of addr */
  TWCR = _BV(TWINT) | _BV(TWEN); /* clear interrupt to start
transmission */
  while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
  switch ((TW_STATUS))
    {
    case TW_MT_DATA_ACK:
      break;

    case TW_MT_DATA_NACK:
      goto quit;

    case TW_MT_ARB_LOST:
      goto begin;

    default:
      goto error;                 /* must send stop condition */
    }

  for (; len > 0; len--)
    {
      TWDR = *buf++;
      TWCR = _BV(TWINT) | _BV(TWEN); /* start transmission */
      while ((TWCR & _BV(TWINT)) == 0) ; /* wait for transmission */
      switch ((TW_STATUS))
        {
        case TW_MT_DATA_NACK:
          goto error;             /* device write protected -- Note [14]
*/

        case TW_MT_DATA_ACK:
          rv++;
          break;

        default:
          goto error;
        }
    }
  quit:
  TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN); /* send stop condition */

  return rv;

  error:
  rv = -1;
  goto quit;
}

int
i2c_write_bytes(uint16_t eeaddr, int len, uint8_t *buf)
{
  int rv, total;

  total = 0;
```

```
  do
    {
#if DEBUG
      printf("Calling i2c_write_page(%d, %d, %p)",
             eeaddr, len, buf);
#endif
      rv = i2c_write_page(eeaddr, len, buf);
#if DEBUG
      printf(" => %d\n", rv);
#endif
      if (rv == -1)
        return -1;
      eeaddr += rv;
      len -= rv;
      buf += rv;
      total += rv;
    }
  while (len > 0);

  return total;
}

int write_register(uint16_t numregister, uint8_t value){
  uint8_t *pvalue;
  int num;
  pvalue = &value;

  num = i2c_write_bytes(numregister, 1, pvalue);

  if(num!=1) return -1;
  else return 1;
}

int read_register(uint16_t numregister){
  int num;
  uint8_t *pvalue;
  uint8_t value = 0;
  pvalue = &value;


  num = i2c_read_bytes(numregister, 1, pvalue);

  if(num!=1) return -1;
  else return (int)*pvalue;
}
```

```
#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <string.h>
#include "usart.h"

char usart_buffer[USART_BUFFER_SIZE];
```

```
volatile unsigned char usart_buffer_pos_first = 0,
usart_buffer_pos_last = 0;
volatile unsigned char usart_buffer_overflow = 0;

void usart_init(unsigned char baud_divider) {

  // Baud rate selection
  UBRRH = 0x00;
  UBRRL = baud_divider;

  // USART setup
  UCSRA = 0x02;           // 0000 0010
                          // U2X enabled
  UCSRC = 0x86;           // 1000 0110
                          // Access UCSRC, Asyncronous 8N1
  UCSRB = 0x98;           // 1001 1000
                          // Receiver enabled, Transmitter enabled
                          // RX Complete interrupt enabled
  sei();                  // Enable interrupts globally
}

void usart_putc(char data) {
    while (!(UCSRA & 0x20)); // Wait untill USART data register is
empty
    // Transmit data
    UDR = data;
}

void usart_puts(char *data) {
  int len, count;

  len = strlen(data);
  for (count = 0; count < len; count++)
    usart_putc(*(data+count));
}

char usart_getc(void) {
  // Wait untill unread data in ring buffer
  if (!usart_buffer_overflow)
    while(usart_buffer_pos_first == usart_buffer_pos_last);
  usart_buffer_overflow = 0;
  // Increase first pointer
  if (++usart_buffer_pos_first >= USART_BUFFER_SIZE)
    usart_buffer_pos_first = 0;
  // Get data from the buffer
  return usart_buffer[usart_buffer_pos_first];
}

unsigned char usart_unread_data(void) {
  if (usart_buffer_overflow)
    return USART_BUFFER_SIZE;
  if (usart_buffer_pos_last > usart_buffer_pos_first)
    return usart_buffer_pos_last - usart_buffer_pos_first;
  if (usart_buffer_pos_last < usart_buffer_pos_first)
    return USART_BUFFER_SIZE-usart_buffer_pos_first
      + usart_buffer_pos_last;
  return 0;
```

```
}

SIGNAL(SIG_UART_RECV) {
  // Increase last buffer
  if (++usart_buffer_pos_last >= USART_BUFFER_SIZE)
    usart_buffer_pos_last = 0;
  if (usart_buffer_pos_first == usart_buffer_pos_last)
    usart_buffer_overflow++;
  // Put data to the buffer
  usart_buffer[usart_buffer_pos_last] = UDR;
}
```